Universidade Federal de Pernambuco

Centro de Informática (CIn)

Graduação em Ciência da Computação

Síntese de arquitetura orientada a serviços a partir de casos de uso.

Victor Hugo Silva do Nascimento

Trabalho de Graduação

Recife, 2016.

Agradecimentos

Dedico todo esse trabalho e minha graduação a meus pais, que me deram todo apoio e suporte durante minha vida. Sem ajuda dos quais não teria sido capaz de ingressar em uma Universidade Federal de referência e concluir o curso de ciências da computação no Centro de Informática. Agradeço a minha irmã que me ajudou bastante nesse trabalho, auxiliando na revisão gramatical e sempre torceu para o meu sucesso.

Amplio os agradecimentos a meus familiares, em especial a minha Tia Adjane e primos que me incentivaram a concluir esse trabalho e me apoiaram com palavras de força em momentos de estresse.

Agradeço aos membros do grupo da disciplina de APS, Iago Caldas Franco de Sá, Joao Guilherme Farias Duda e Tiago Nogueira, pela elaboração conjunta do projeto SharingShopList, usado como estudo de caso neste trabalho.

Agradeço também ao meu orientador, professor Augusto Sampaio, por estar sempre disponível para responder minhas dúvidas e pelo suporte dado na elaboração deste trabalho, mesmo em seu período de férias.

Por fim, sou grato aos professores do Centro de Informática, que em sua maioria exercem seu trabalho com excelência e se empenham em dar uma formação de qualidade aos alunos do centro.

Resumo

Arquitetura Orientada a Serviços (*Service Oriented Architecture* – SOA) é um estilo arquitetural que se consolidando na academia e na indústria e tem como objetivo possibilitar o desenvolvimento de aplicações mais flexíveis, modulares e reutilizáveis. Alinhando esse estilo arquitetural com o Desenvolvimento Dirigido a Modelos, em que sistemas completos podem ser construídos automaticamente a partir de transformações automáticas de modelos, foi desenvolvido um processo sistemático de projeto arquitetural orientado a serviços por Braga [6].

Apesar de se ter um processo já elaborado e validado, existe pouco apoio ferramental para colocá-lo em prática. O objetivo deste trabalho é realizar um estudo no processo, identificando tarefas ou etapas que possam ser automatizadas através de um sistema de software e implementar uma ferramenta para agilizar sua utilização. Um estudo de caso com um projeto SOA é utilizado para exemplificar a automatização feita no processo e evidenciar suas vantagens.

Sumário

Capitulo 1 – Introdução

O processo de desenvolvimento de softwares é custoso e complexo, envolvendo um conjunto de atividades realizadas por um grupo de pessoas, visando a criação e a manutenção de uma nova aplicação. Nos Estados Unidos são investidos anualmente U$250 bilhões no desenvolvimento de aproximadamente 175.000 projetos de T.I., onde apenas 16.2% são concluídos dentro do prazo e dos custos inicialmente estimados [13]. Os motivos de falha são diversos, 52.7% dos projetos falham em estimar custos e chegam a custar até 189% da estimativa original [13].

No entanto, o ciclo de vida do software não se encerra com o término do desenvolvimento. Após concluído e entregue, o sistema tende a passar por diversas mudanças, sendo necessário evoluí-lo. O custo de desenvolvimento, apesar de ser alto, ainda é menor do que custa para manter o sistema. A facilidade e a acessibilidade da manutenção são diretamente impactadas pelo processo de desenvolvimento, que pode ocasionar em um aumento de até 75% no custo da manutenção, caso não tenha sido levado em consideração a evolução do sistema em sua elaboração [14].

Dentre as atividades envolvidas no processo de desenvolvimento, a elaboração da arquitetura do software é a que tem relacionamento direto com a manutenção do sistema e consiste em organizar os componentes do software a fim de criar um sistema modular e flexível, ou seja, fácil de evoluir e com custo reduzido de manutenção. Dentre os diversos padrões e estilos arquiteturais existentes, a Arquitetura Orientada a Serviços (SOA – do inglês *Service Oriented Architecture*) é uma abordagem que vem ganhando bastante atenção de pesquisadores da área, bem como da indústria de TI. SOA organiza os componentes do sistema em Serviços, que são funções de negócios bem definidas, autocontidas e independentes entre si, permitindo reuso, escalabilidade e flexibilidade [6]. O padrão de camadas e o Movel View Controller (MVC), e suas variações, são outros exemplos de padrões de arquitetura. Porém o foco do trabalho é em SOA.

O trabalho desenvolvido por Braga [6] sugere um processo de análise e projeto arquitetural de software sistemático, independente de ferramentas, de tecnologias e orientado a serviços. Esse processo possui três fases básicas e recebe como entrada alguns artefatos comuns aos processos de desenvolvimento tradicionais, como

documento de requisitos e diagrama de casos de uso. A partir desses, é criada, sistematicamente, uma arquitetura SOA.

Este trabalho de graduação tem como objetivo implementar uma ferramenta de apoio a algumas tarefas desse processo e, em especial, a transformação de diagramas de casos de uso em um diagrama que descreve a arquitetura de serviços. Este é um passo importante para automatizar o processo de geração de uma arquitetura SOA. O usuário poderá importar o diagrama de casos de uso, elaborado em uma ferramenta CASE, e a ferramenta proposta gera um diagrama de arquitetura de serviços em SoaML (SOA Modeling Language), como um artefato da fase de Análise. Na sequência, esse diagrama é refinado em um diagrama de componentes, no nível de projeto.

No capítulo 2, são explicados conceitos fundamentais para o entendimento do trabalho, como: definição de arquitetura de software e sua importância, padrões arquiteturais e arquitetura orientada a serviços. O capítulo 3 resume o processo de elaboração de SOA descrito detalhadamente em [6]. Em seguida, no capítulo 4, são apresentadas as tecnologias utilizadas, a ferramenta desenvolvida. Um estudo de caso para ilustrar as facilidades oferecidas pela ferramenta proposta é apresentado no Capítulo 5. Por fim, no capítulo 6, o trabalho é concluído, onde apresentamos, também, alguns trabalhos relacionados e tópicos para trabalhos futuros.

Capítulo 2 – Conceitos e Fundamentos

Um dos principais desafios durante a execução do processo de desenvolvimento de softwares é a elaboração da arquitetura do sistema. O conceito de arquitetura de software não possui uma definição precisa, mas pode ser entendido como uma visão abstrata do sistema, que não leva em consideração detalhes de implementação e nem representação de dados, concentrando-se apenas no comportamento e interação entre seus elementos [1].

Essa representação abstrata permite aos *stakeholders* (usuário final, desenvolvedores, engenheiro de testes, gerentes de projetos, entre outros) um entendimento de alto nível, possibilitando uma fácil comunicação, análise e validação do sistema. A arquitetura impacta diretamente cada *stakeholder*, pois é possível analisar e inferir várias características, como por exemplo: disponibilidade, confiabilidade, custo e tempo de desenvolvimento (a partir da análise de cada módulo ou elemento) [1]. Quanto maior e mais complexo for o sistema, maior a necessidade de ter essa facilidade de comunicação que impacta diretamente no sucesso do projeto de software.

Fowler [2] conceitua arquitetura de software como um conjunto de decisões tomadas no início do projeto, difíceis de serem mudadas e que definem a estrutura de componentes do sistema e suas interações. O autor explicita outra vantagem de se ter uma arquitetura bem definida: a possibilidade de tomar decisões de projeto o mais cedo possível no processo de desenvolvimento, permitindo a validação e análise dos impactos, distribuição e manutenção do sistema [1].

Mudanças arquiteturais realizadas tardiamente no desenvolvimento são bastante custosas, atrasando o desenvolvimento do sistema. Portanto é importante entender quais impactos a arquitetura tem em um software, alguns deles são:

- Restrições na implementação: a implementação do sistema deve seguir as decisões estruturais descritas na arquitetura, preservando o relacionamento entre os elementos definidos e garantindo o funcionamento da responsabilidade que cada elemento assume com os demais. Essa modularização facilita o desenvolvimento, pois é possível

implementar diversos elementos do sistema paralelamente, uma vez que se conhece como devem interagir e a responsabilidade de cada componente.

- Definição de atributos de qualidade do sistema: Atributos de qualidade como performance, escalabilidade, possibilidade de reuso (via baixo acoplamento e alta coesão) e segurança devem ser considerados na definição da arquitetura do sistema. Em [1], são descritas técnicas que possibilitam prever se as decisões arquiteturais tomadas garantem as qualidades desejadas baseando-se apenas na arquitetura. Porém é importante ressaltar que, apesar de ser um passo necessário para definir tais características, garantir as funcionalidades dos atributos de qualidade depende de cada tarefa no processo de desenvolvimento.
- Facilidade de análise e gerenciamento de mudanças: a maior parte do custo no ciclo de vida do software acontece depois de seu desenvolvimento, portanto mudanças são inevitáveis. Ter conhecimento profundo da arquitetura do sistema é um requisito essencial para gerenciar mudanças, que consiste em decidir quando é necessário avaliar suas consequências e determinar como realizá-las de forma que o risco seja o menor possível. Em [1] são definidas três categorias de mudanças:
    - Local: ocorre quando um único elemento precisa ser modificado.
    - Não-local: ocorre quando vários elementos precisam ser modificados, porém não são feitas mudanças arquiteturais.
    - Arquitetural: ocorre quando há uma mudança na interação dos elementos do sistema que provavelmente trará consequências para todo o sistema.

  Portanto, uma arquitetura desejável é aquela em que as mudanças necessárias são fáceis de serem realizadas, podendo ser local ou não-local. Se uma mudança arquitetural for necessária significa que ocorreu uma falha na tomada de decisões no projeto do sistema.

Apesar da importância e das vantagens associadas à arquitetura, não existe uma notação ou terminologia única para descrevê-la. Engenheiros de software costumam

utilizar regras e estilos que surgiram informalmente com o tempo, alguns são bem documentados e utilizados na indústria, como a UML [3] que tornou-se um padrão da OMG [3].

2.1 Padrões Arquiteturais

Existem problemas que são encontrados com frequência em diversas áreas de estudo; a técnica mais comum para resolvê-los é procurar problemas similares que já foram solucionados e aplicar a essência da solução encontrada no que se quer resolver. Padrões propõem soluções genéricas, reutilizáveis e adaptáveis para cada tipo de problema. O termo padrão é definido em [5] como a relação entre um problema, o contexto onde está inserido e sua solução.

Uma das categorias de padrões definidos por [4] são os arquiteturais que descrevem a organização estrutural fundamental para os sistemas de software. Padrões arquiteturais definem um conjunto de subsistemas ou elementos, suas responsabilidades e regras de como devem interagir. Conhecê-los permite entender melhor o processo de elaboração arquitetural, além de serem soluções reutilizáveis e de acelerarem essa tarefa.

Padrões são geralmente descritos da seguinte forma:

- Nome do padrão
- Contexto
- Problema
- Solução
    - Estrutura
    - Dinâmica
    - Consequências

Utilizando essa estrutura, alguns padrões são apresentados a fim de exemplificar os conceitos já definidos.

2.1.1 Arquitetura em camadas

Esse padrão permite estrutura a aplicação em camadas, onde cada uma representa um nível de abstração do sistema e interage apenas com as camadas vizinhas [3].

- Contexto: uma aplicação complexa o suficiente que requer decomposição [4].
- Problema: desenvolver um sistema complexo que realiza operações em diversos níveis, onde as de níveis mais alto dependem de operações de níveis mais baixos. Por exemplo: um sistema que recebe dados de usuário realiza algum processamento sobre os dados, se comunica com sistemas externos e utiliza sistemas de bancos de dados para persistência.
- Solução: é possível identificar a diferença de nível entre as operações do exemplo, o fluxo de informação e suas dependências. O padrão de camadas orienta estruturar a aplicação em camadas, começando da camada de menor nível de abstração e incrementando verticalmente com camadas de maior nível de abstração [4]. Ou seja, camadas de maior nível são colocadas acima de camadas de menor abstração, formando uma pilha de camadas. Portanto, uma camada N provê serviços à camada N+1 e depende dos serviços providos pela camada N-1. Uma arquitetura em camadas para o exemplo mencionado a cima poderia ser:

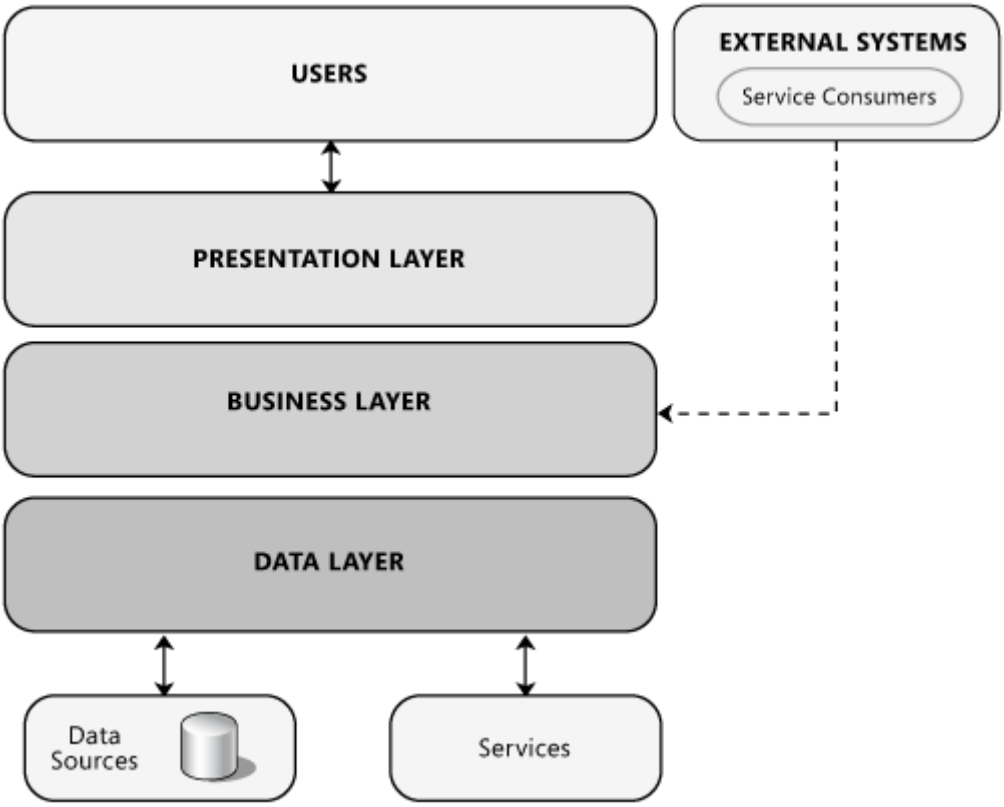


Figura 2.1 – Diagrama de arquitetura em camadas [7].

O usuário interage diretamente com a camada de apresentação, responsável por receber e apresentar dados ao usuário. A segunda camada é responsável pelas regras de negócio do sistema, que se comunica com sistemas externos. E por último, a camada de dados é responsável por persistir os dados necessários.

Estrutura: uma camada pode ser descrita conforme mostra a figura 2.2.

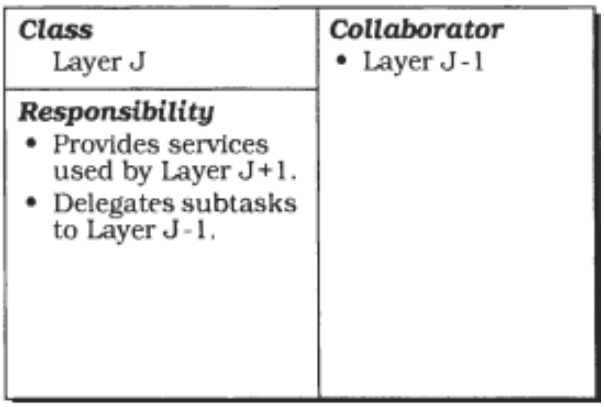


Figura 2.2 – Representação de uma camada em cartão CRC [4].

É colocado o nome da camada, suas responsabilidades e quem colabora com a camada. O colaborador é a camada logo abaixo que provê serviços a ela. Uma camada pode ser composta por subsistemas e componentes complexos; é importante que os elementos que a compõe funcionem no mesmo nível de abstração. Considere o modelo definido pela figura 2.3.

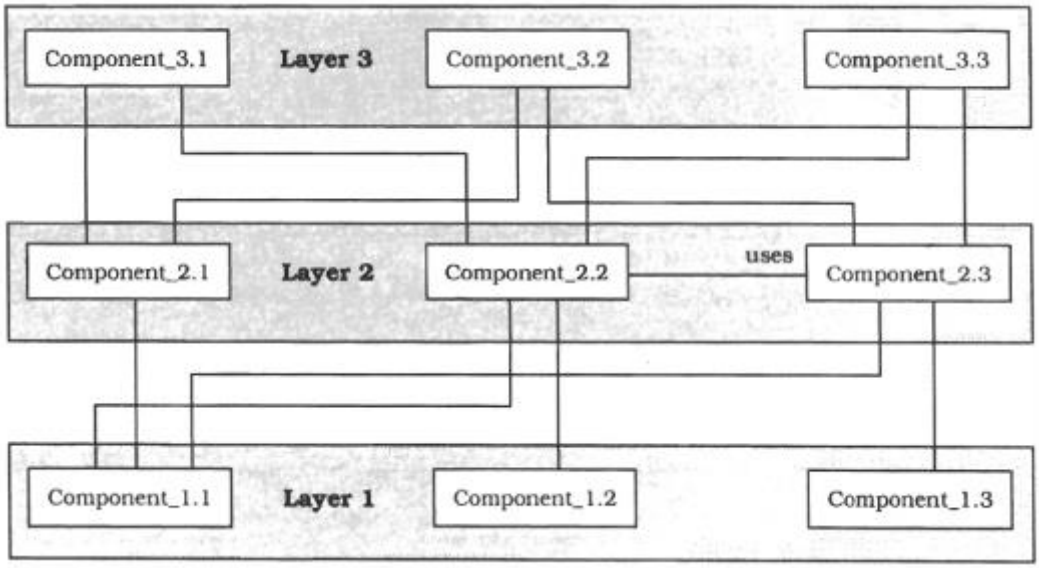


Figura 2.3 – Componentes de uma camada [4].

- Neste exemplo, cada camada possui três componentes, que podem depender de outros da mesma camada ou de camadas vizinhas, mas nunca de outras que não sejam vizinhas. Existe outra variação em que cada camada deve ter uma interface, e todo acesso é feito através desta, não podendo ocorrer uso direto de elementos de outras camadas, como ocorre no exemplo ("*Componente_3.*2" utilizando diretamente o "*Componente_2.1*" e "*Componente_2.3*").
- Dinâmica: O cliente faz requisições à camada de maior abstração, que como não pode lidar com a requisição sozinho, delega subtarefas à camada logo abaixo; este processo ocorre até a camada de menor abstração. A resposta da requisição é então retornada até a camada de maior abstração que a apresenta ao cliente.

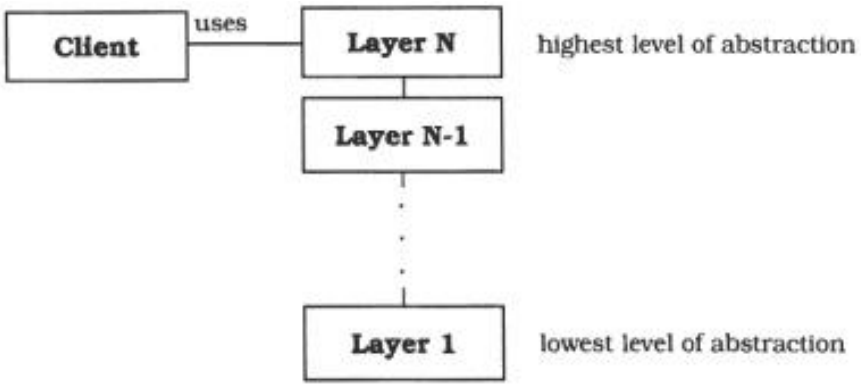


Figura 2.4 – Comunicação de camadas [4].

- Consequências: Alguns benefícios em utilizar arquitetura em camadas são:
    - Se as camadas representarem abstrações e tiverem interfaces bem definidas, é possível reutiliza-las em outros contextos. Em [4], mostra-se que existem estudos empíricos que sugerem o reuso de camadas, como componentes caixa preta, para diminuir o esforço de desenvolvimento e número de defeitos.
    - Mudanças feitas em alguma camada não afetam o resto do sistema, caso o contrato (interface) que a camada deve cumprir não seja alterado, afetando apenas a camada modificada. Isso contribui para a portabilidade, facilita a manutenção e testes no sistema, já que é possível testar cada camada independente dos outros componentes.
    - Níveis de abstração bem definidos permitem trocar implementações da mesma interface com mais facilidade, contribuindo para o uso de diferentes produtos de terceiros. Por exemplo: uma camada responsável por persistir dados pode utilizar SGBDs (Sistemas de gerenciamento de banco de dados) de diversos tipos e empresas (MySQL, MongoDB, etc), desde que a interface da camada seja mantida.

  Porém, também existem algumas desvantagens:

    - Se o comportamento (interface) da camada em si precisar ser alterado, isso causará um efeito cascata que afeta todo o sistema. Esse tipo de mudança, caso necessária, poderá implicar em mudanças em várias outras camadas.

- Estabelecer a granularidade correta das camadas de um sistema pode ser uma decisão difícil e por isso uma desvantagem. Arquiteturas com poucas camadas não exploram bem o potencial do padrão e dificultam o reuso, portabilidade e facilidade de mudança. Porém, camadas em excesso aumentam a complexidade desnecessariamente, podendo comprometer a performance do sistema.
- Todo padrão adiciona uma certa complexidade para prover as vantagens descritas, e isso impacta na eficiência e performance do sistema [4].

2.1.2 Model-View-Controller (MVC)

Divide o sistema em três componentes básicos: modelos (*model),* visões (*view)* e controladores (*controllers*).

- Contexto: Desenvolver aplicações interativas com interface usuário-máquina flexível [4].
- Problema: Necessidade de mudar dinamicamente a GUI (interface gráfica do usuário, do inglês *graphical user interface*), que pode variar entre diferentes tipos de usuário ou contexto em que requisições são feitas. Se a implementação das regras de negócios da aplicação estiver fortemente acoplada com código de interface de usuário, atender este requisito pode ser custoso e sujeito a erros.
- Solução: O objetivo do MVC é separar os dados e regras de negócio da aplicação de como são apresentados ao usuário. Esse objetivo é atingido através da divisão do sistema em três módulos:
  - Model - Representa os dados e regras de negócio da aplicação. É independente de como são representados.
  - View - Responsável por mostrar informações, obtidas do *model*, ao usuário (GUI). Sendo possível implementar diferentes *visões* para o *model*.
  - Controller - Responsável por fazer a ponte de comunicação entre o *model* e a *view*. Em uma aplicação que segue o padrão MVC, o usuário interage com o *controller*, que ao receber requisições repassa para o *model* que retorna informações. O *controller* então envia informações atualizadas

para os objetos da *view*. Existem algumas variações do MVC que definem essa dinâmica de forma diferente. Em [8] são apresentas algumas variações. No *supervising controller*, por exemplo, qualquer modificação no *model* propaga a mudança em todas as *views* associadas, utilizando o padrão de projeto *observer* [8].

Através dessa modularização é possível dissociar completamente as visões dos modelos, de forma que qualquer alteração feita neles seja refletida em todas as *views* dependentes.

- Estrutura: A estrutura é relativamente simples, consistindo apenas da apresentação dos três módulos.

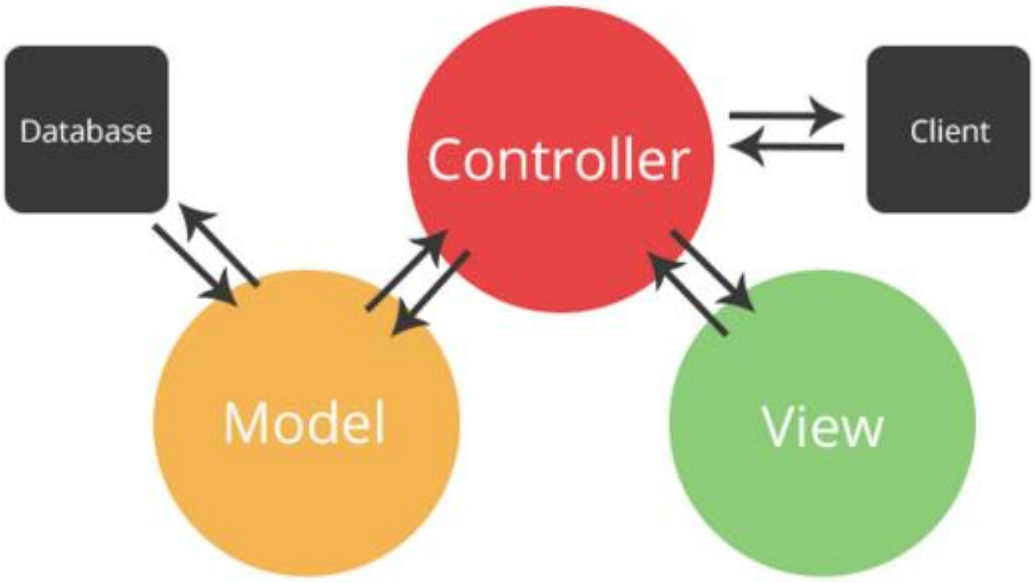


Figura 2.5- Padrão arquitetural MVC [8].

O diagrama da figura 2.5 exemplifica o MVC tradicional, onde o *controller* é responsável por todo fluxo de comunicação entre o *model* e a *view*.

- Dinâmica: Na versão original, o *controller* recebe requisições do usuário e as redireciona para o *model.* Esse faz o processamento necessário e retorna informações para o *controller*, que carrega a *view* correta. O diagrama de sequência da figura 2.6 exemplifica esse fluxo no contexto de uma aplicação web que recebe requisições http.

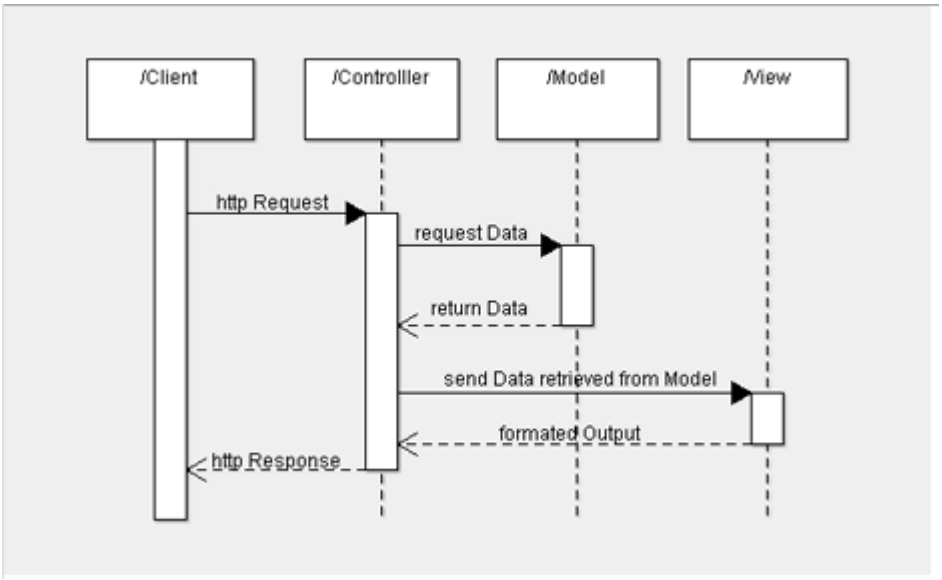


Figura 2.6 – Diagrama de Sequência do MVC [4].

Na variação *supervising controller*, modificações no *model* são propagadas para as *views*, conforme o diagrama de sequência exibido na figura 2.7.

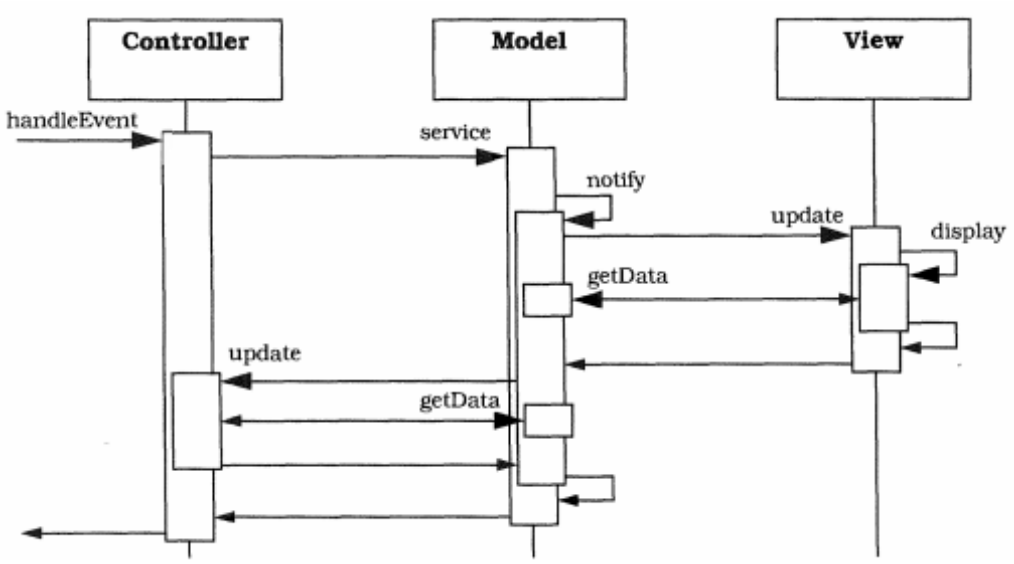


Figura 2.7 – Diagrama de sequência *supervising controller [4].*

Embora pareça que o *model* interaja diretamente com a *view*, o que criaria uma dependência, isso não acontece de forma direta devido ao padrão de projetos *observer* [4]. Portanto os mesmos benefícios da versão original são preservados nessa variação.

- Consequências: Alguns benefícios de se utilizar o padrão MVC são:
    - Separação de regras de negócio e dados (*model*) da interface do usuário (*view*), o que permite implementar várias visões para o mesmo *model*, alcançando assim flexibilidade na apresentação de dados até em tempo de execução.
    - Atualização das diferentes visões podem ser feitas de maneira automática, através de *observers*, e dinâmica (em tempo de execução).
    - Modelo independente de código de interface permite maior facilidade na portabilidade para outras plataformas. Não sendo necessário fazer modificações em regras de negócio, apenas no código das visões.

  Algumas desvantagens de se utilizar MVC são:

    - Aumento de complexidade na construção de interfaces simples, pois cada interface necessita ser implementada através de um par *view* e *controller* [4].
    - Toda lógica referente à interface de usuário está presente na *view* e em seu *controller*, incluindo dependências com a plataforma em que se desenvolve. Por exemplo, em um sistema web a interface com o usuário depende de tecnologias *front-end* como html, javascript e css; portar um sistema web para uma plataforma que use tecnologias diferentes, vai exigir identificar a parte do código da *view* e *controller* que possui essas dependências e reescreve-lo [4].

Esses dois padrões arquiteturais são amplamente utilizados e servem como base para várias soluções e frameworks encontrados no mercado. Um estilo arquitetural que vem se consolidando na literatura e sendo progressivamente adotado no mercado é a arquitetura orientada a serviços (SOA, do inglês *Service Oriented Architecture*) [10].

2.2 Arquitetura Orientada a Serviço

Uma das estratégias para resolução de problemas complexos é decompô-lo em partes que representam uma porção específica do problema. Desta forma é possível construir a solução desejada de maneira incremental a partir da integração das partes decompostas. Esta estratégia é a utilizada em diversos padrões e estilos arquiteturas, inclusive SOA. No entanto, seu diferencial é como essa decomposição é feita.

SOA propõe decompor o sistema em serviços, definidos como: uma função de negócio independente que recebe e responde a requisições através de uma interface bem definida [12]. Portanto, cada serviço encapsula uma parte lógica do sistema e são independentes entre si, o que permite serem distribuídos e reutilizáveis em diferentes contextos. No entanto, [11] aponta que apesar da independência é necessário que serviços se adequem a algumas regras e convenções comuns, que padronizam aspectos chaves de cada negócio, sem criar um acoplamento muito forte entre eles. Por exemplo, em um sistema de lojas pode ser necessário definir uma moeda comum de negócio. Os serviços existem de forma autônoma, mas não isolados um dos outros [11].

Serviços são constituídos de dois componentes [10]:

- Interface do serviço – descreve a identificação do serviço, definição dos parâmetros e convenções para responder a requisições do consumidor. Ou seja, descreve o que o serviço oferece.
- A implementação do serviço – código de implementação das funções descritas em sua interface. Descreve como o serviço realiza suas operações.

Para consumir ou prover um serviço é necessário ter conhecimento apenas de sua interface, já que funcionam como uma caixa-preta, e podem ser de três tipos [6]:

- **Serviço de Entidade:** Derivado de uma ou mais entidades de negócio, possuindo um alto grau de reutilização. Geralmente são serviços que fazem operações CRUD (Create, *Read*, *Update* e *Delete*).
- **Serviço de Tarefa:** Tipo de serviço mais específico que possui baixo grau de reuso. Consome outros serviços para atender seus consumidores.

- **Serviço de Utilidade:** Tem alto grau de reuso, pois implementa funcionalidades comuns a vários tipos de aplicações, como, por exemplo: log, notificação, transformação de informações.

A figura 2.8 exemplifica como esses três tipos de serviço se relacionam [6].

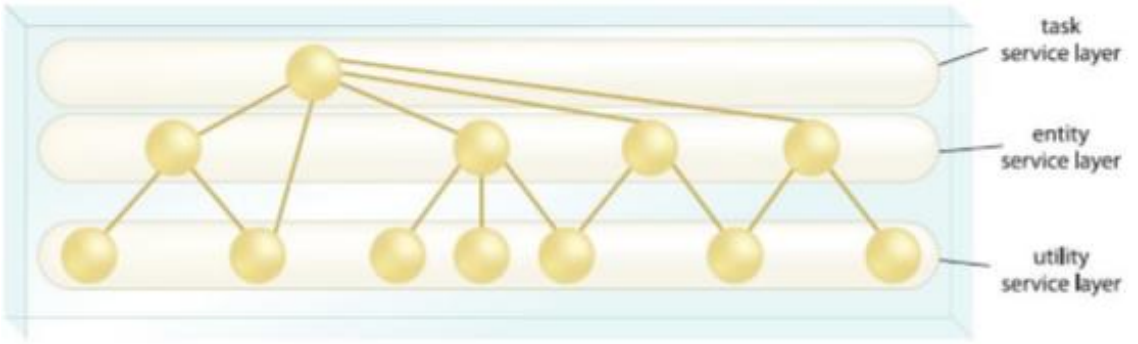


Figura 2.8 – Relacionamento entre os tipos de serviços [6].

A arquitetura do sistema é construída a partir da composição de serviços, que podem ter sido implementados do zero ou reutilizados de outros sistemas.

2.2.1 Vantagens e Desvantagens

Em [10] são apontadas as seguintes vantagens de se utilizar SOA:

- Baixo acoplamento de componentes (serviços), o que permite o reuso e facilidade em adicionar ou atualizar serviços existentes.
- Facilidade na integração e interoperabilidade entre serviços, o que aumenta a eficiência no uso de recursos disponíveis.
- Melhor escalabilidade e possibilidade de desenvolver aplicações independentemente e em paralelo.
- Redução nos custos de desenvolvimento do sistema, devido ao reuso de componentes.

Existem algumas desvantagens na utilização de SOA em um ambiente de desenvolvimento.

- Um grande investimento inicial é necessário, e o retorno pode demorar a acontecer [6].

- Assim como ocorre no padrão arquitetural de camadas, definir a granularidade dos serviços é uma tarefa difícil e crucial para o sucesso da utilização do padrão. Serviços muitos gerais podem causar problemas de reuso e não proporcionam as vantagens que o SOA promete.
- Integração com outros sistemas pode ser difícil, pois geralmente tecnologias de integração são proprietárias, sendo necessário se adequar àquela tecnologia específica, dificultando o reuso, devido a dependência existente com a tecnologia proprietária [10].

2.2.2 SOAML

Por ser um campo de estudo emergente e promissor [9], SoaML (*Service Oriented Architecture Modeling Language*) foi especificado pela OMG para descrever e projetar sistemas SOA através de diagramas UML, permitindo modelar requisitos, especificar serviços de sistemas, interface de serviços e projetar os serviços internamente [6].

Os serviços são representados através de um contrato (*interface*) que define como as requisições devem ser feitas (restrições) e a resposta retornada. Serviços são consumidos e/ou oferecidos por participantes, que podem ser: pessoas, organizações ou outros sistemas.

Os participantes possuem dois tipos de portas de interação com serviços, oferecidos através de portas com estereótipo <<*Service*>> e consumidos através de portas com estereótipo <<*Request*>>. O uso de serviços e participantes é exemplificado na figura 2.9.

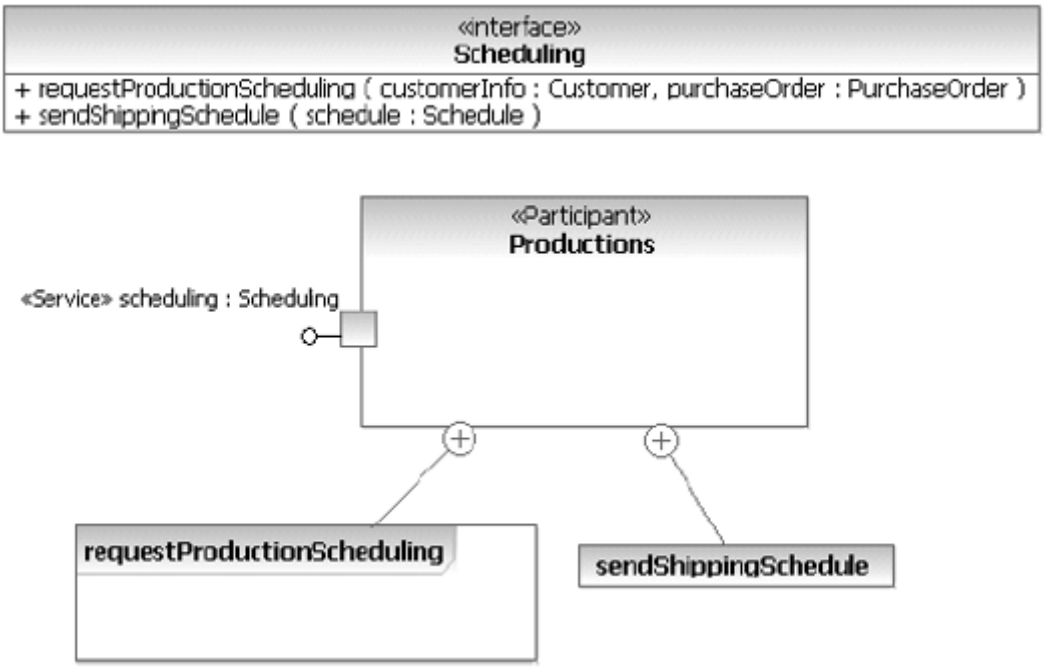


Figura 2.9 – uso de participantes e serviço [6]

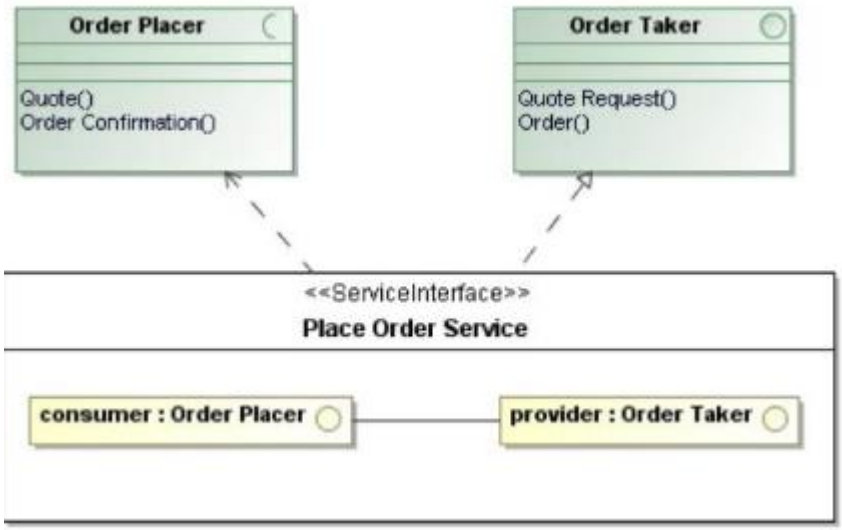
Na figura, o serviço *scheduling* é definido através de uma interface que descreve duas requisições, *sendShippingSchedule* e *requestProductionSchedulling*. O participante *Productions* oferece o serviço definido através de uma porta, que é descrita usando a seguinte nomenclatura: "<<Nome Estereotipo>> nome: Tipo". O estereotipo <<Service>> indica que o serviço está sendo ofertado pelo participante e possui o tipo da interface definida *Scheduling*.

Existem ainda os seguintes tipos de contrato de serviço:

- Interface Simples (*Simple Interface)* – Interface utilizada quando não é necessário conhecer o consumidor do serviço, muitas vezes usada para expor diretamente as capacidades do sistema ou para definir serviços mais simples que não tem protocolos [6]. É, portanto, feita uma interação de mão única, o consumidor chama operações do fornecedor e recebe respostas (ou não), o

fornecedor não conhece quem está consumindo seus serviços. A interface definida no exemplo da figura X.X é uma *Simple Interface*.

- Interface de Serviço (*Service Interface*) – Descreve serviços bidirecionais, onde o fornecedor tem conhecimento do consumidor e pode invocá-lo durante a execução das operações. É definida em termos do provedor de serviço e especifica a interface ofertada e, se for o caso, a que espera consumir [6]. É utilizada uma classe UML para definir esse tipo de interface e pode ser também utilizado um diagrama de sequência para descrever o fluxo dos serviços ofertados.



Figuro 2.10 – Exemplo de interface de serviço [6].

A interface de serviço *Place Order Service* definida na figura 2.10, especifica duas interfaces simples: *Order Placer* o serviço que será consumido (utilizada); e *Order Taker* o serviço ofertado. Dentro da classe são definidos os participantes, consumidor e fornecedor, e seus papéis na interação do serviço.

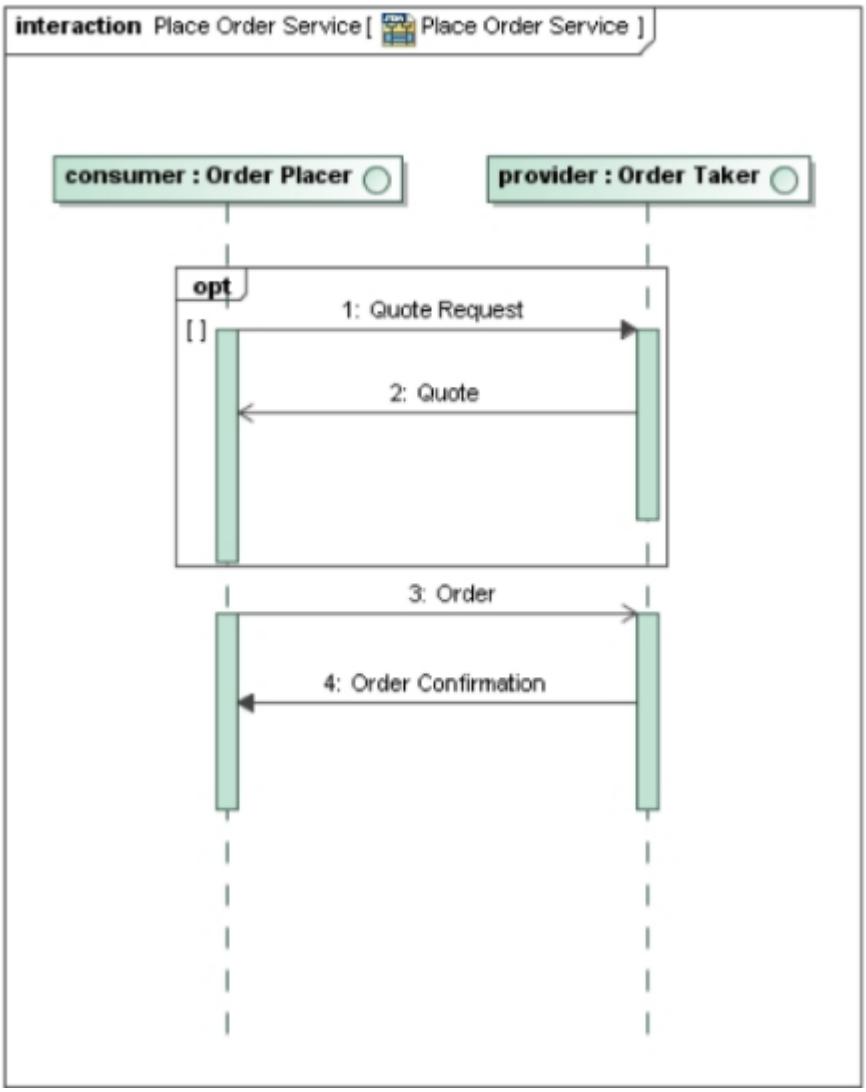


Figura 2.11 – Exemplo de fluxo de operações de interface de serviço[6].

O diagrama de sequência no exemplo acima, mostra como ocorre o fluxo das operações definidas na interface de serviço.

SoaML permite visualizar a colaboração de serviços em um sistema, com clareza, através do diagrama de arquitetura de serviço, que contém um conjunto de participantes, com papeis bem definidos, fornecendo e consumindo serviços. Pode ser descrita em dois níveis: arquitetura de domínio e arquitetura dos participantes [6]. O

primeiro descreve como os participantes de um domínio específico trabalham em conjunto para atingir seus objetivos, descrito utilizando um diagrama UML de colaboração. Já o segundo nível, define a arquitetura de serviço interna de um participante [6], descrito utilizando um diagrama UML de classe ou componentes.

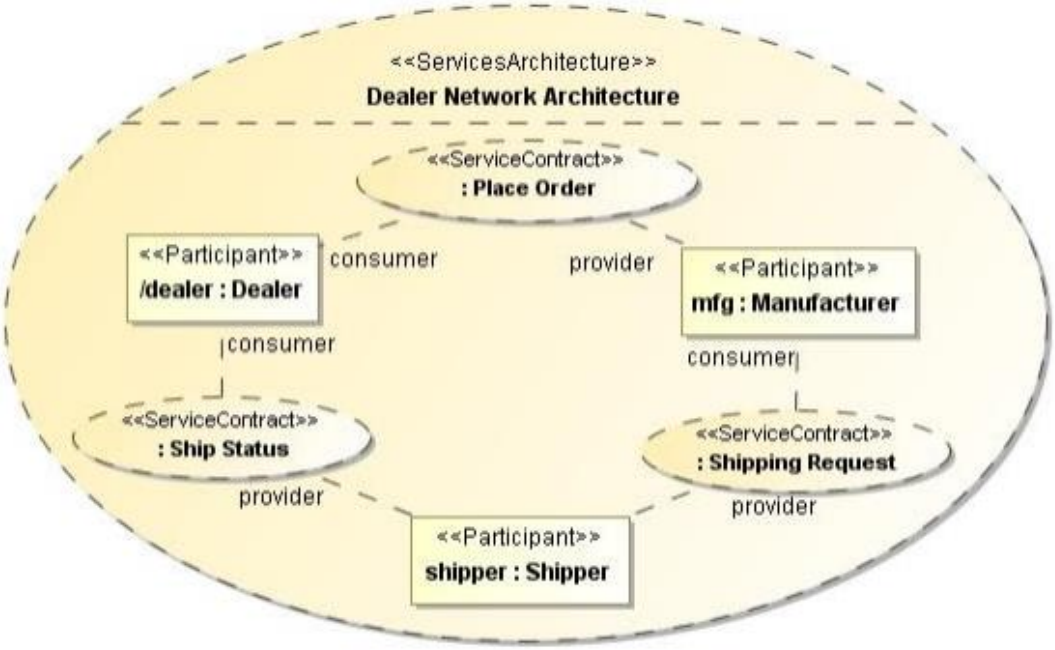


Figura 2.12 – Exemplo Diagrama Arquitetura de Serviço [6].

A figura 2.12 descreve a arquitetura de uma *Dealer Network,* onde são definidos os participantes e serviços:

- *Shipper* – que oferta os serviços *Shipping Request* e *Ship Status*;
- *Manufacturer* – consome o serviço *Shipping Request* e oferece o serviço *Place Order*.
- *Dealer* – Consome os serviços *Ship Status* e *Place Order*.

A partir do diagrama também é possível inferir a dinâmica de interação dos participantes. O *dealer* faz um pedido ao *Manufacturer (*através do serviço *Place Order),* que por sua vez confirma o pedido e solicita o envio da mercadoria (serviço *Shipping Request)* ao participante *Shipper,* responsável por enviar o produto. É possível verificar o status do envio através do serviço *Ship Status.*

Capítulo 3 – Processo elaboração SOA

Em [6], Braga define um processo sistemático de projeto arquitetural de software dirigido a modelo e orientado a serviços, objetivando projetar arquiteturas estáveis, flexíveis e modulares e resolver problemas comuns encontrados em projeto de softwares, como, por exemplo: acoplamento entre módulos do sistema (back-*end* e *front-end*), integração de aplicações e melhorar o entendimento dos *stakeholders* sobre o sistema.4

O processo consiste na realização de três atividades principais: Especificar Modelo de Negócio, Analisar Serviços e Projetar Serviços, conforme demonstra a figura 3.1.

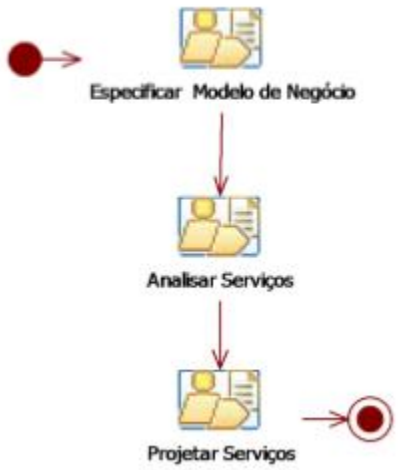


Figura 3.1 – Tarefas do processo de projeto SOA [6].

A primeira atividade consiste em elaborar artefatos para compreensão dos *stakeholders*; são chamados de modelos independentes de computação (CIM) [6] e são elaborados por especialistas do domínio. Na atividade seguinte, Analisar Serviços, utiliza-se SoaML para gerar artefatos de maior grau de abstração e independentes de plataforma e tecnologia. Por último, na atividade de Projetar Serviço, são gerados modelos considerando-se tecnologias, plataformas e linguagens de programação. A seguir, será apresentada, em mais detalhes, cada uma dessas atividades e os artefatos gerados.

3.1 Especificar Modelo de Negócio

Esta atividade recebe como entrada os seguintes artefatos: Documento de requisitos, Casos de Uso, Processo de modelo de negócio (BPM), Histórias de usuário, entre outros, gerando artefatos que possam ser facilmente compreendidos por todos os *stakeholders* envolvidos no projeto, proporcionando um melhor entendimento do sistema. É subdivido em três passos, conforme mostra a figura 3.2.

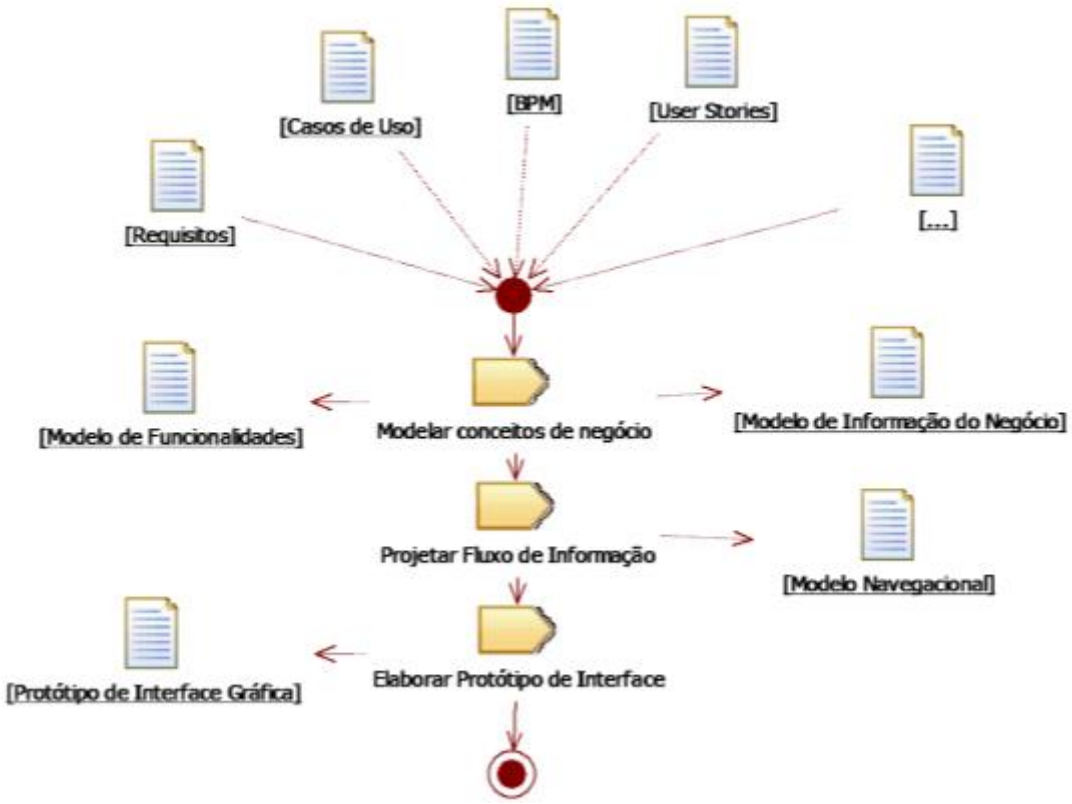


Figura 3.2 – Especificar Modelo de Negócio [6].

- Modelar conceitos de negócio - Recebe os artefatos citados anteriormente, que especificam o problema, as necessidades do cliente e funcionalidades do sistema. A partir desses, é criado o Modelo de Informação do Negócio (MIN) e o Modelo de Funcionalidades (MF). O primeiro tem como objetivo capturar conceitos do domínio e suas relações, ajudando a capturar os conceitos de negócio que o sistema irá manipular. É representado por um diagrama de classes UML sem atributos e operações contendo apenas classes conceituais do sistema [6]. Já o MF, é um diagrama de casos de uso agrupados em pacotes, casos de uso presentes em um mesmo pacote possuem funcionalidades comuns e muito provavelmente farão

parte de um mesmo serviço. A figura 3.3 e 3.4 mostram um exemplo do MIN e do MF, respectivamente.

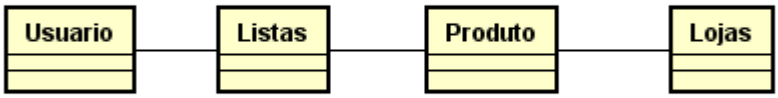


Figura 3.3 – Modelo de Informação de Negócio

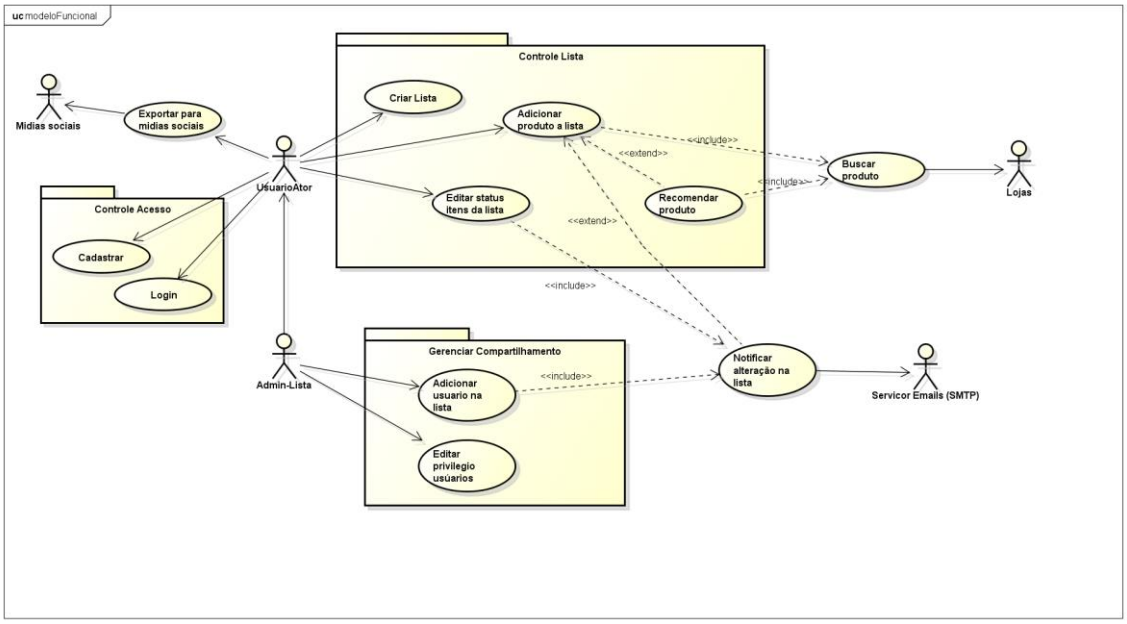


Figura 3.4 – Modelo de Funcionalidades.

- Projetar Fluxo de Informação - O próximo passo do fluxo recebe como entrada o MIN e gera o Modelo Navegacional (MN), que consiste em um diagrama de classes UML representando o fluxo das telas do sistema. A figura 3.5 exemplifica um MN.

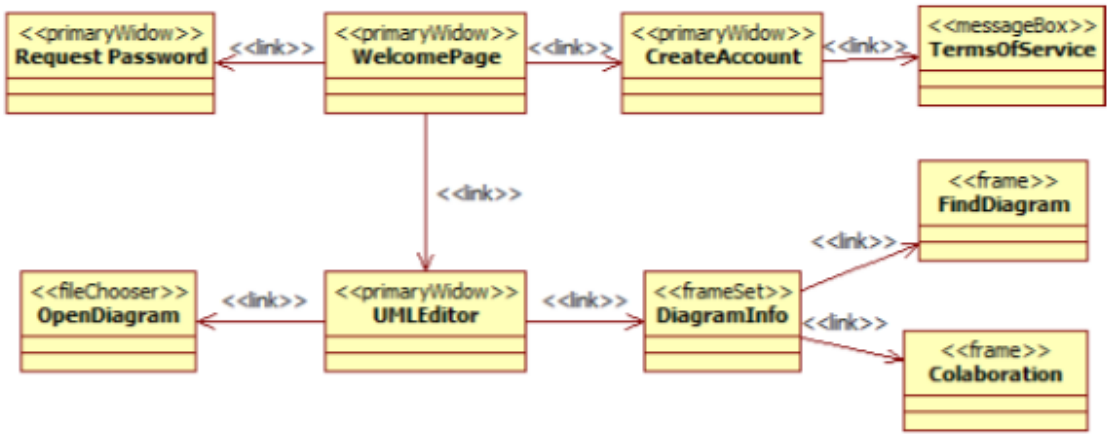


Figura 3.5 – Modelo Navegacional do sistema

Cada classe UML representa um elemento da interface gráfica e as setas os links entre elas.

- Elaborar Protótipo de Interface - A última atividade deste fluxo tem como objetivo criar o Protótipo de Interface Gráfica (PIG), que é um *layout* completo do sistema, contendo todas as funcionalidades, telas, conteúdo e mensagens do sistema. A figura 3.6 mostra um exemplo de um PIG.

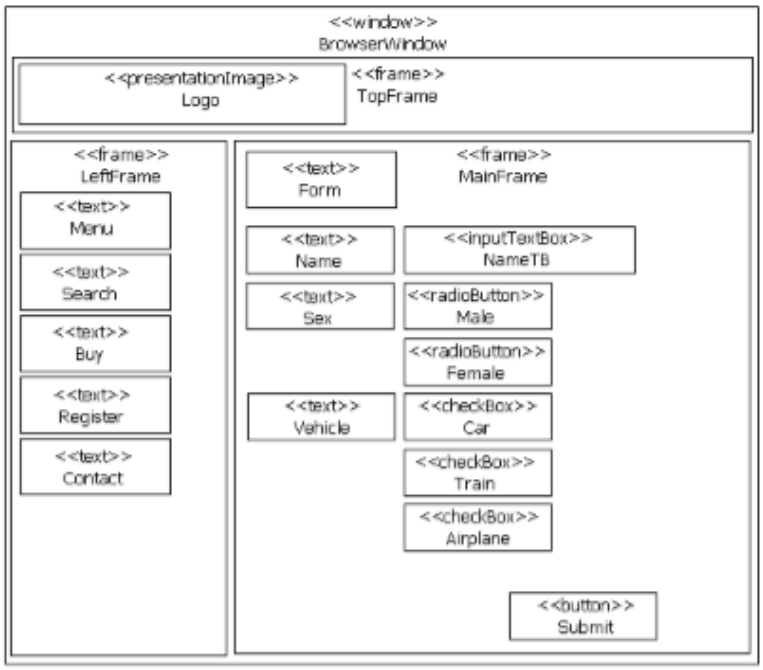


Figura 3.6 – Exemplo de Protótipo de Interface Gráfica (PIG) [6].

3.2 Analisar Serviços

O objetivo desta atividade é criar artefatos que já conferem uma visão arquitetural do sistema. Esta atividade possui três passos: Identificar Serviços, Refinar Serviços e Identificar Componentes.

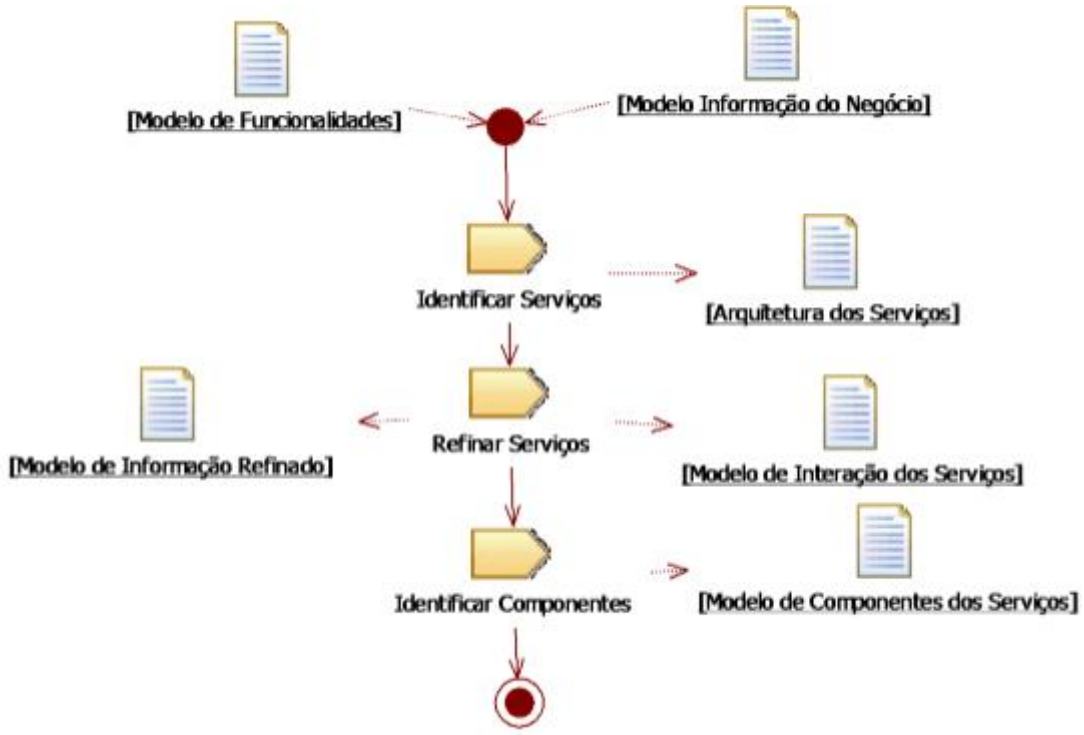


Figura 3.7 – Atividades e artefatos de Analisar Serviços [6].

- Identificar Serviços – Recebe como entrada o MIN e o MF, gerando a Arquitetura de Serviços (AS), que descreve como os participantes trabalham em conjunto para fornecer e utilizar os serviços descritos nos contratos de serviço [6]. A AS é representada por um diagrama de classe UML, sem atributos, onde cada classe possui os estereótipos <<Service Contract>> ou <<Participant>>. É criada a partir do MF, seguindo as seguintes regras:
    - Cada pacote contendo casos de uso são mapeados em um contrato de serviço. Casos de uso que não estejam contidos em um pacote também geram contratos de serviço.
    - Cada ator é mapeado em um participante e, um participante independente que representa o sistema é criado. Os demais participantes devem consumir

apenas serviços ofertados pelo Sistema, e esse consome serviços oferecidos por outros participantes.

Essas regras são exemplificadas na figura 3.8.

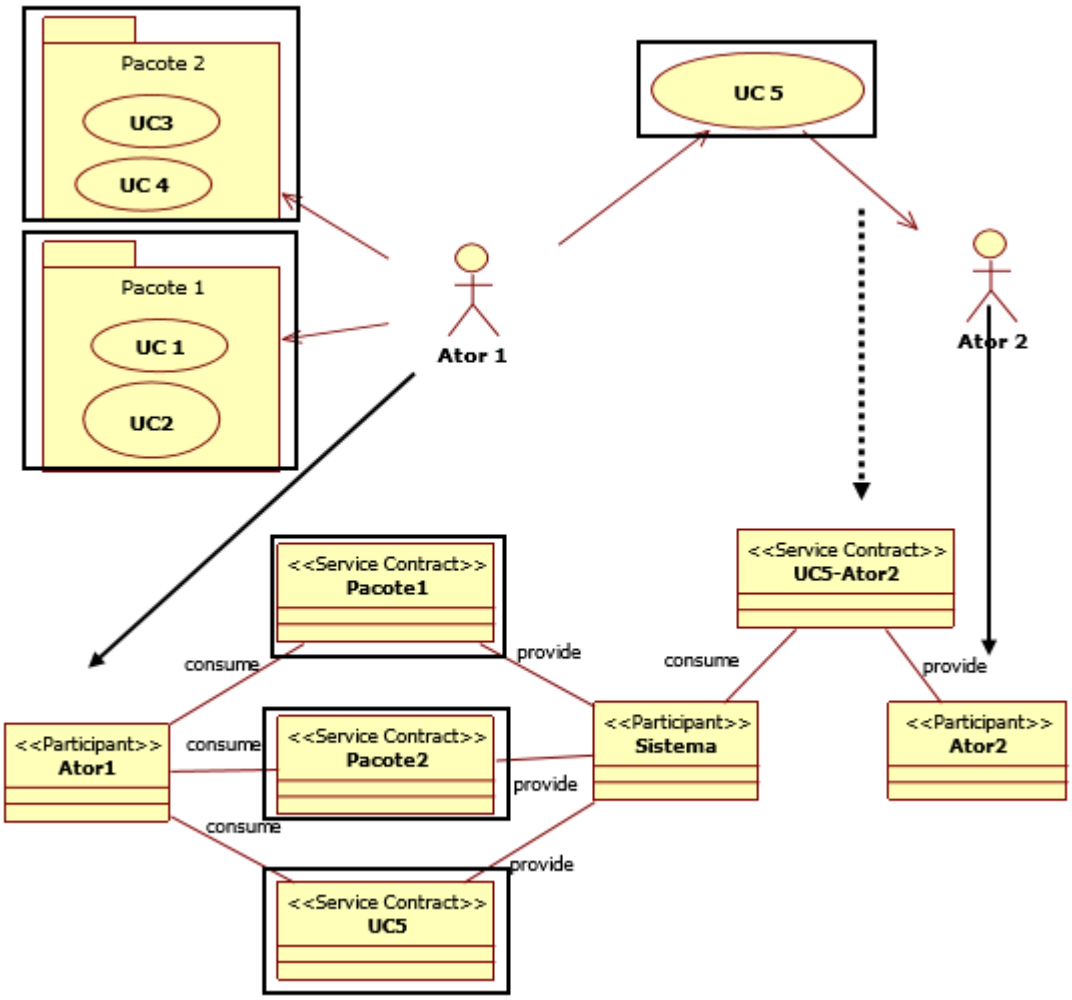


Figura 3.8 – Mapeamento MF em diagrama de Arquitetura de Serviço[6].

Os pacotes 1 e 2 geraram os contratos de serviço “Pacote1” e “Pacote2”, respectivamente. O caso de uso UC5, que não está incluso em nenhum pacote, gera um contrato independente. Cada ator é mapeado em um participante, e a direção da relação entre o caso de uso e o ator indica se o ator consome ou oferece o serviço. O participante Sistema é criado e consome o serviço UC05, ofertado pelo participante “Ator2”. É importante notar que o Sistema oferta todos os serviços consumidos pelos demais participantes.

- Refinar Serviços – Tem como objetivo criar o Modelo de Interação de Serviços (MIS) e o MIN Refinado, a partir do AS e o MIN. A partir desses modelos é possível identificar as funções especificas em que os serviços podem ser invocados. São as operações das interfaces do serviço. O primeiro passo é identificar os serviços de entidade, isso é feito a partir dos <<*entity service*>> presentes no MIN, conforme mostra o exemplo da figura 3.9.

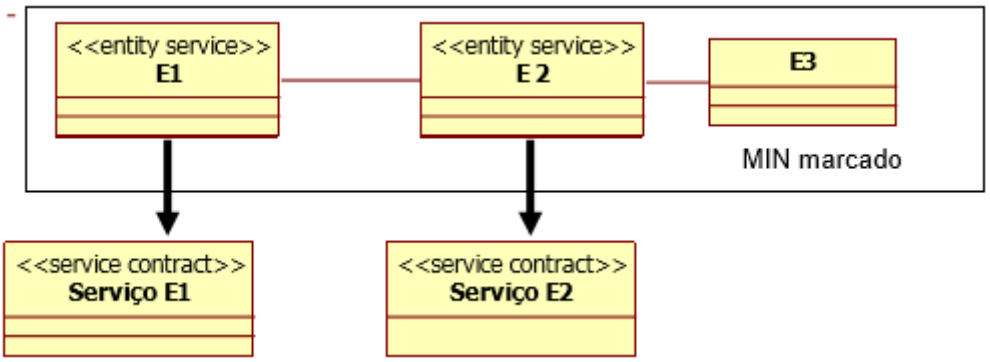


Figura 3.9 – Serviços de entidade [6].

A partir dos serviços de entidade e da arquitetura de serviço, é possível criar o MIS seguindo os passos abaixo:

1. A partir dos participantes consumidores, criam-se interfaces;
2. Interfaces também são criadas com o nome dos contratos de serviço e serviços de entidade;
3. Para cada contrato de serviço é criado um diagrama de sequência, contendo: o consumidor do serviço, o próprio contrato de serviço e os serviços de entidades utilizados.

Um *template* do MIS é exemplificado na figura 3.10.

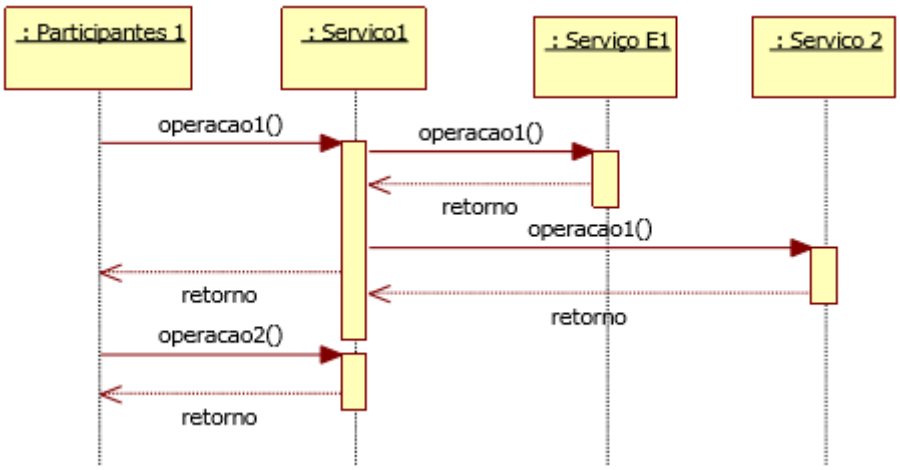


Figura 3.10 – Modelo de Interação de Serviço [6].

Durante a elaboração do MIS pode ser identificado a necessidade de criar novas entidades e validar as entidades já criadas, ocasionando na atualização do MIN.

- Identificar Componentes – O objetivo dessa tarefa é criar o Modelo de Componentes dos Serviços (MCS), construído a partir do AS e do MIS, obedecendo as seguintes regras:
    - Os participantes exclusivamente consumidores são mapeados em componentes;
    - Os contratos de serviços são mapeados em componentes com interfaces de acesso;
    - São descritas as operações das interfaces dos componentes, essas são extraídas do MIS.

A figura 3.11 exemplifica a criação do MCS.

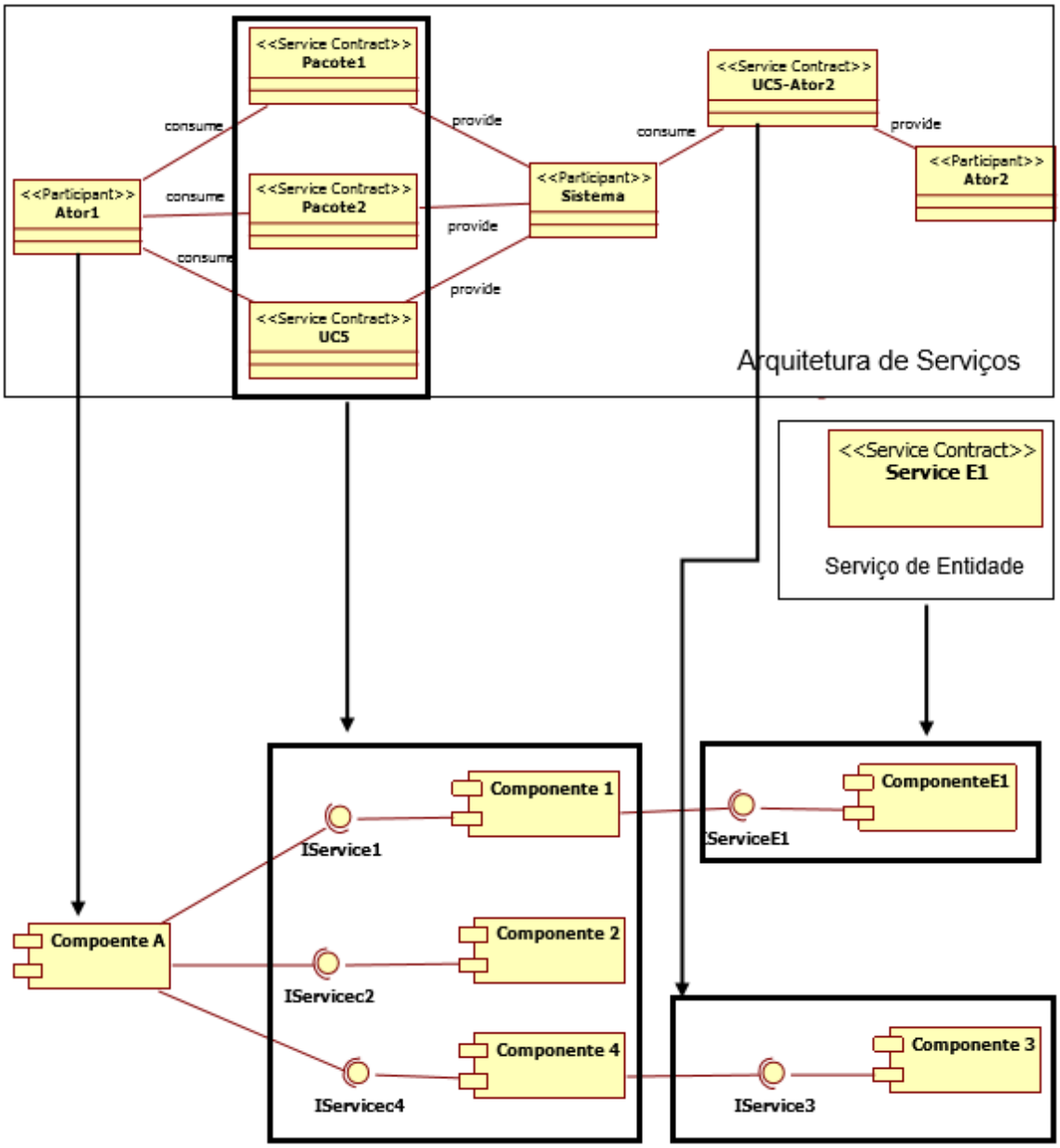


Figura 3.11 – Modelo de Componente de Serviços [6].

O “Ator1” é mapeado no “Componente A” e cada contrato de serviço é mapeado em um componente com interface de acesso.

3.3 Projetar Serviços

Nesta última etapa do processo é levado em consideração detalhes de implementação, como: plataforma, linguagem de programação, etc. Uma arquitetura mais detalhada é desenvolvida e novas oportunidades de reuso são identificadas. A figura 3.12 demonstra o fluxo de atividades dessa fase.

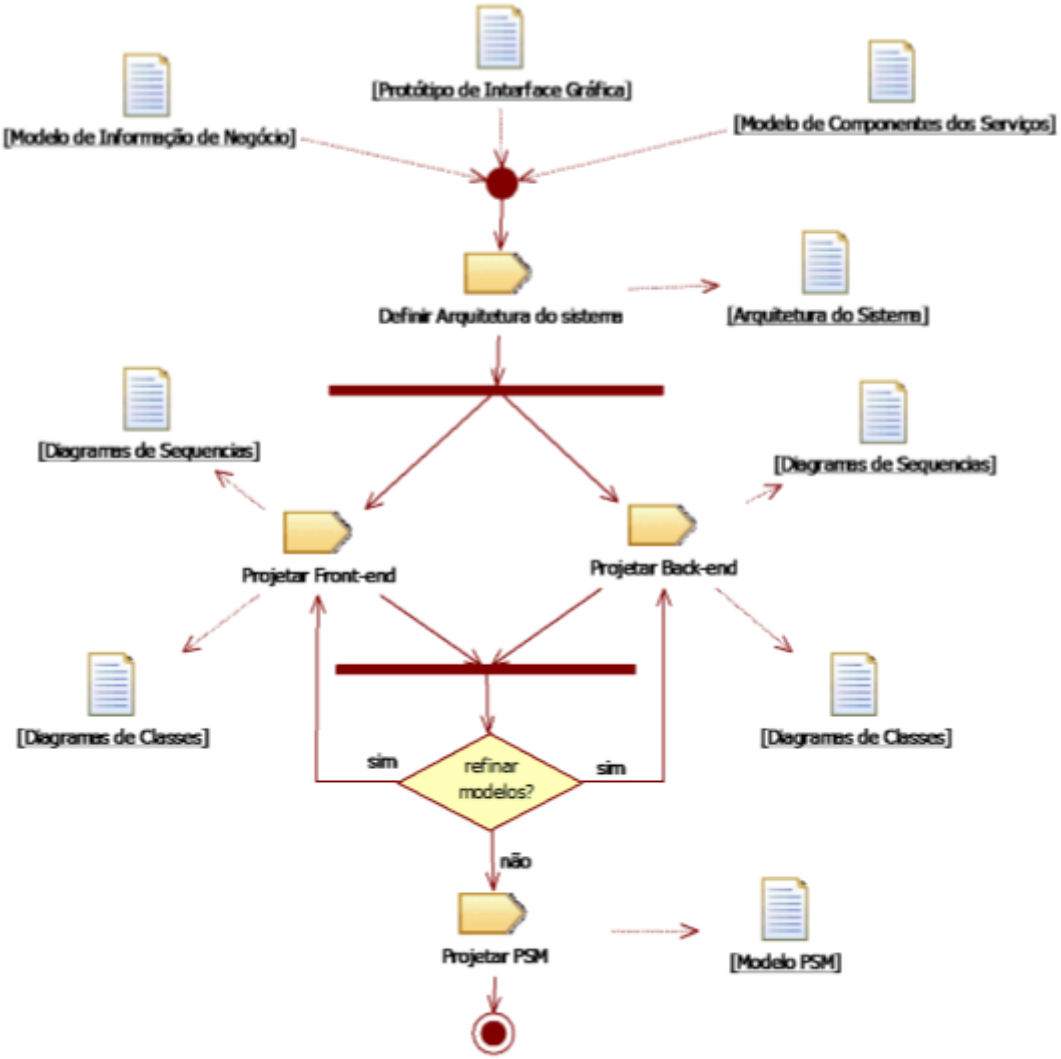


Figura 3.12 - fluxo de atividades Projetar Serviços [6].

- Definir Arquitetura do sistema – O primeiro passo do fluxo, recebe o PIG, MIN e o MCS, tendo como saída a Arquitetura do Sistema, uma evolução da Arquitetura de Componentes de Serviços contendo todos os elementos necessários para modelar os componentes internamente; os componentes *front-end* são marcados com o estereotipo <<*Front-end*>>. A figura 3.13 exemplifica o diagrama de Arquitetura de Sistema da fase de projeto.

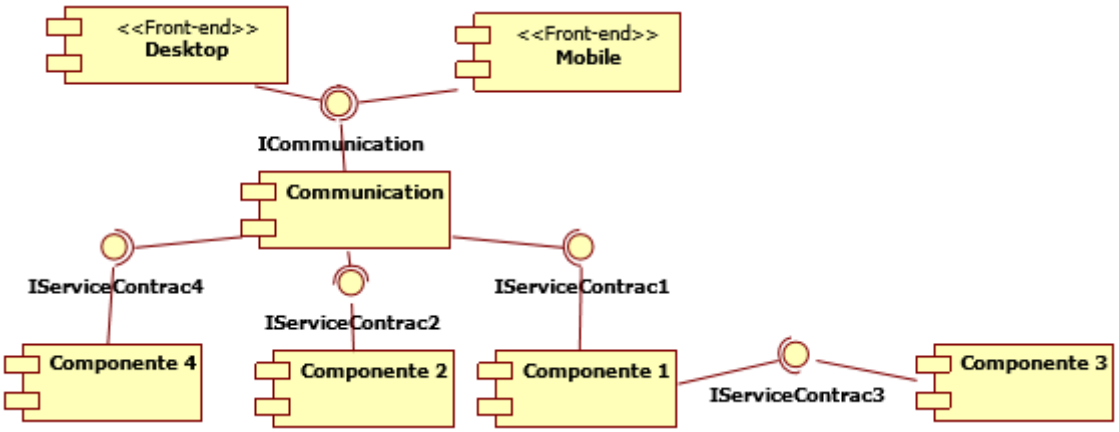


Figura 3.13 – Diagrama de Arquitetura de Sistema.

É importante notar que o *front-end* é separado do *back-end* por uma interface; devido a isso, as duas próximas atividades podem ser desenvolvidas em paralelo. Os modelos gerados na etapa anterior (Analisar Serviços) são revisados e refinados, onde são definidas quais tecnologias e *frameworks* serão utilizados no projeto do sistema.

- Projetar *back-end* – Obedecendo as regras e padrões definidos anteriormente, cada componente *back-end* é projetado. Ou seja, é construído o diagrama de classe e de sequência de cada componente sendo necessário detalhar atributos, operações e relacionamentos das classes projetadas. Durante o projeto dos componentes, mudanças podem ocorrer o que ocasionará na necessidade de atualizar o MIN. Nessa etapa também é feito o projeto de banco de dados baseando-se nas tecnologias escolhidas.
- Projetar *front-end* – Neste passo constrói-se diagramas de sequência e de classe, utilizando-se como base o Protótipo de Interface Gráfica, a integração entre o *front-end* e *back-end,* os guias e padrões definidos para cada componente *front-end* do sistema [6].

Capítulo 4 – Automatização do processo

Este trabalho tem como objetivo automatizar passos da tarefa “analisar serviço” apresentada na seção 2.3. Inicialmente, é gerado o diagrama de arquitetura de serviço a partir do modelo funcional, dado como entrada pelo usuário. Posteriormente, é possível gerar o modelo de componentes dos serviços, a partir da arquitetura de serviço e modelo do modelo de interação de serviços.

A automatização dessas tarefas é possível, pois as transformações realizadas seguem regras bem definidas e invariáveis. A figura 4.1 demonstra a conversão de um modelo de funcionalidades em uma arquitetura de serviço.

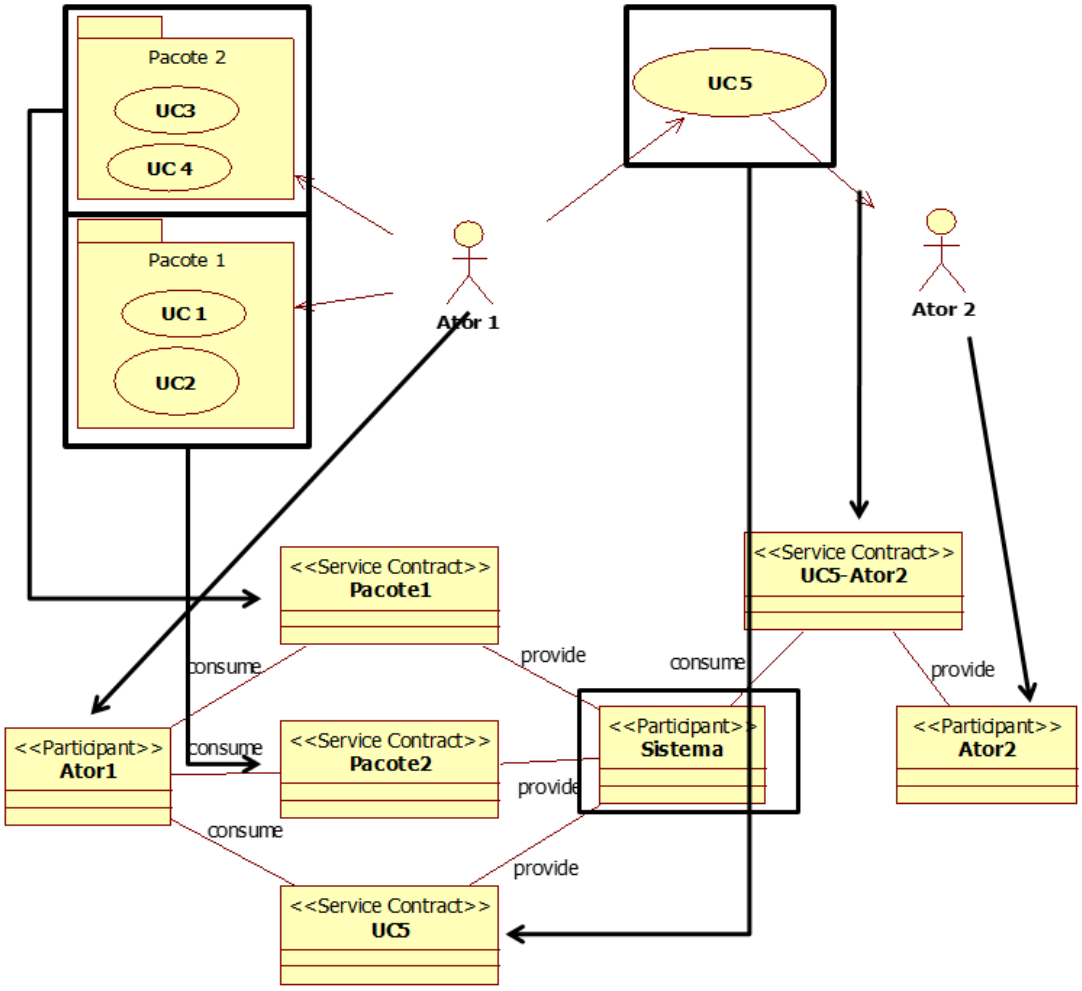


Figura 4.1 – Transformação MF em AS [15].

Cada ator é mapeado em uma classe UML com estereótipo <<Participant>>, indicando que é uma entidade que consome ou oferece serviços. Cada pacote ou caso de uso isolado é mapeado em uma classe UML com estereótipo <<Service Contract>>,

entidades que representam serviços consumidos e ofertados por participantes. É criado o participante sistema, que oferece todos os serviços identificados, consumindo serviços ofertados por participantes para oferecê-los a outros. Ou seja, todo os serviços consumidos por outros participantes devem ser ofertados pelo participante Sistema. A figura 4.2 mostra a transformação do diagrama de arquitetura de serviço no modelo de componentes do sistema.

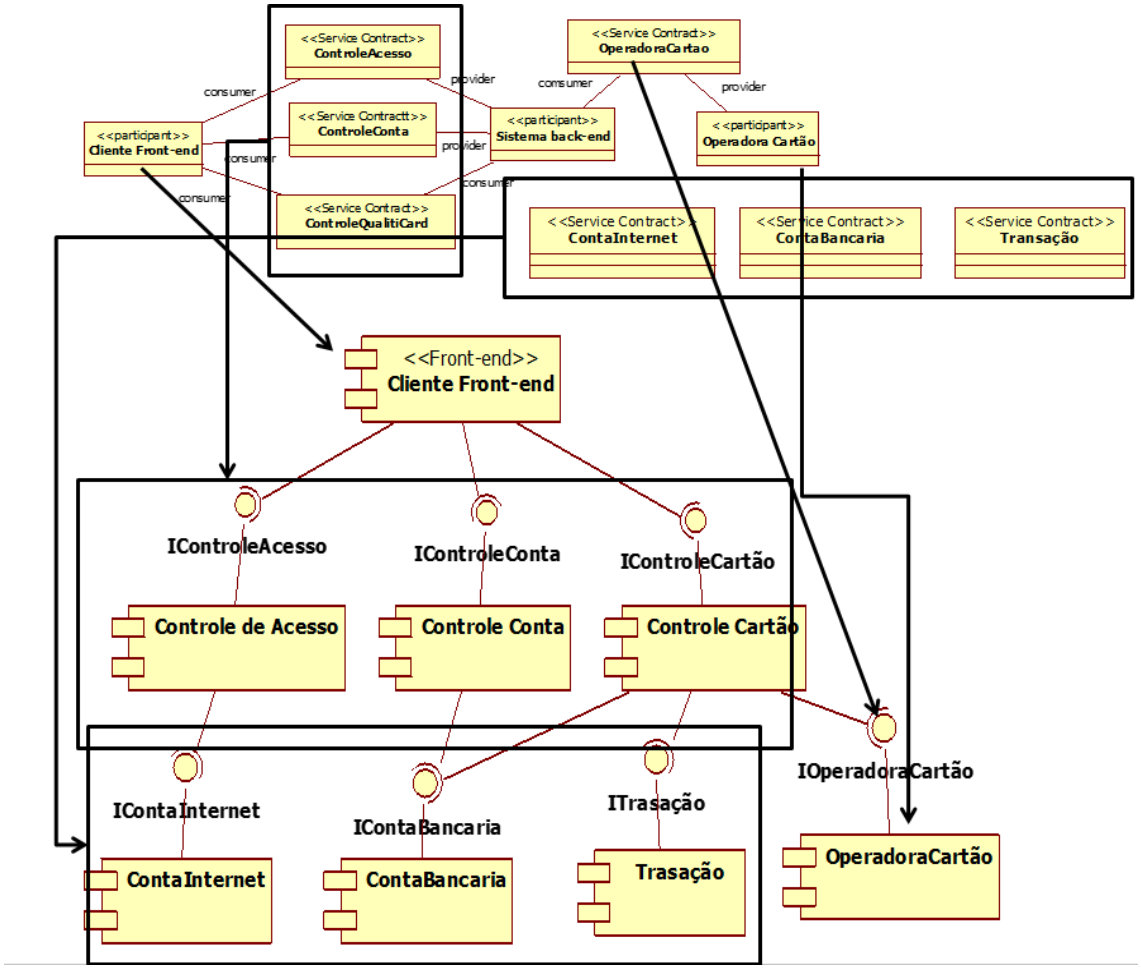


Figura 4.2 – Transformação Arquitetura de Serviço em Modelo de Componentes [15].

Cada serviço é mapeado em um componente com uma interface de acesso e o participante torna-se o componente *front-end* do sistema. É importante notar que existem serviços não conectados no diagrama de arquitetura de serviço do exemplo (figura 4.2), esses são serviços de entidades extraídos do modelo de interação de serviços, mostrado na figura 4.3.

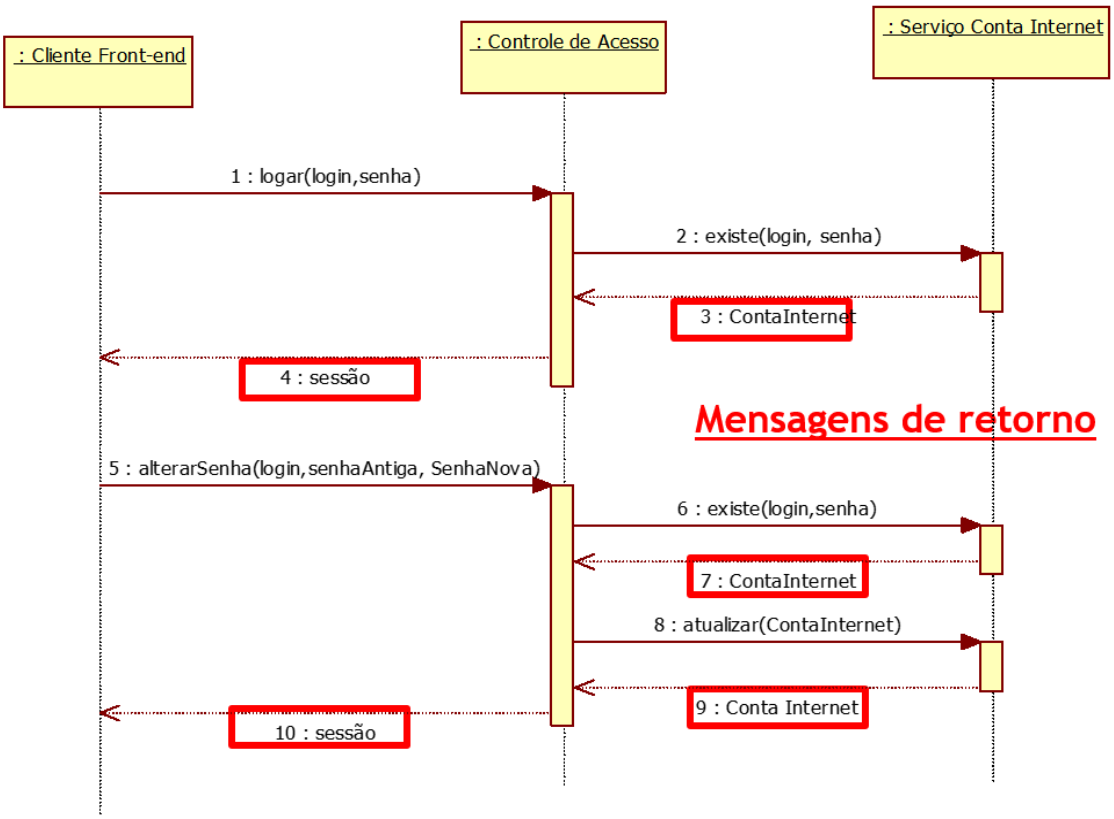


Figura 4.3 – Modelo de interação de serviços [15].

A ferramenta foi desenvolvida como um sistema web, em que o usuário faz o *upload* dos diagramas necessários em formato XML (do inglês *Extensible Markup Language*). Existem várias ferramentas de modelagem UML e cada uma possui uma representação XML diferente para seus diagramas. Na versão inicial do sistema desenvolvido foi escolhido oferecer suporte a ferramenta CASE Astah, pois possibilita a criação do modelo de funcionalidades com facilidade e é possível exportar diagramas em XML na sua versão gratuita.

4.1 Tecnologias

As principais tecnologias utilizadas no desenvolvimento da solução proposta em uma plataforma web foram NodeJs e TypeScript.

4.1.1 NodeJs

NodeJs é um ambiente de execução multiplataforma para desenvolvimento e execução de aplicações web do lado do servidor [16]. Aplicações NodeJs são escritas utilizando a linguagem Javascript, já muito utilizada no desenvolvimento de *front-end*

de sistemas web, sendo executado por browsers. Ou seja, usando NodeJs é possível implementar o *back-end* de aplicações web utilizando a mesma linguagem do *front-end.* Apesar de ser recente, desenvolvido em 2009 por Ryan Dahl, adquiriu grande aceitação da comunidade por ser uma plataforma *open-source,* registrado sob licença MIT, possuindo mais de 200 mil módulos desenvolvidos pela comunidade e sendo um dos projetos com mais contribuições no repositório GitHub [17].

O diferencial do Node é sua arquitetura orientada a eventos e operações de entrada e saída (I/O) não bloqueantes, o que permite construir aplicações mais escaláveis [16]. Em aplicações tradicionais, operações de I/O são bloqueantes, ou seja, a execução do programa é bloqueada até o término desse tipo de operação. Em aplicações web, operações de entrada e saída ocorrem o tempo todo, e consomem muitos recursos, pois são realizadas por diversos usuários simultaneamente, portanto não podem ser bloqueantes.

Uma solução muito comum para esse problema é o uso de *threads*, uma parte do processo que compartilha memória com outras *threads* do mesmo processo, enquanto uma *thread* realiza uma operação de I/O, outras continuam com o processamento das requisições. O principal problema dessa abordagem é que *threads* precisam ser sincronizadas, já que compartilham acesso a memória, o que pode gerar inconsistência de dados (caso haja escrita). Sincronizar *threads* pode ser um trabalho custoso, dependendo da complexidade da aplicação, além de forçar os desenvolvedores a prever todo tipo de interação com a memória compartilhada pelas diversas *threads* [16].

Node utiliza uma arquitetura orientada a eventos e com uma única *thread* capaz de atender a um alto número de requisições e permite realizar operações de entrada e saída não bloqueantes, implementado através de um mecanismo chamado de *event loop* demonstrado na figura 4.4.

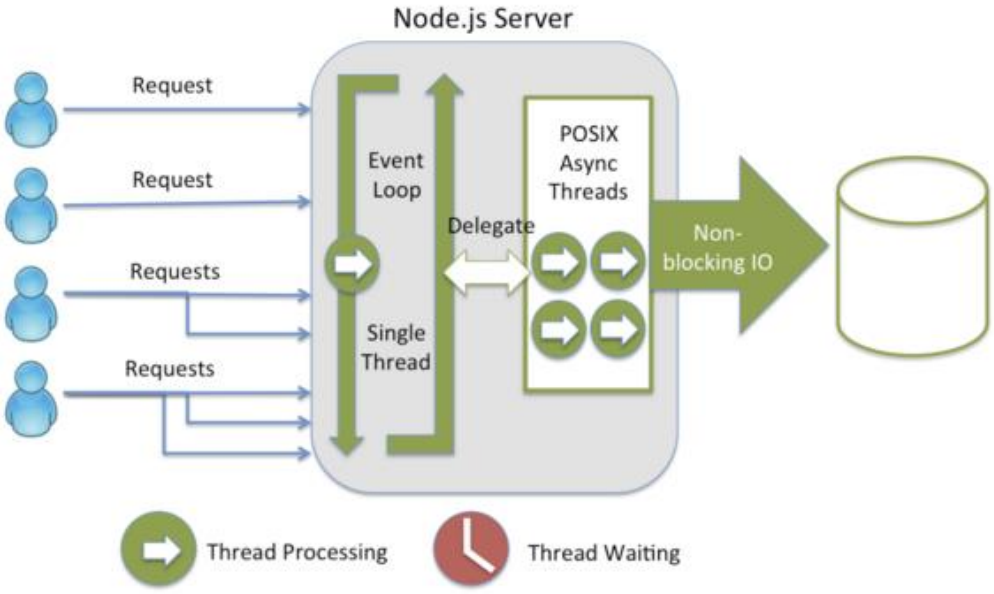


Figura 4.4 – Diagrama arquitetura NodeJs [19].

O *event loop* se baseia fortemente no paradigma de programação orientada a eventos e funções chamadas de *callbacks*. Basicamente ao iniciar o servidor, Node instancia todas as variáveis e funções necessárias e espera que eventos ocorram. Quando um evento é disparado, a função de *callback* registrado para aquele evento é chamada e sua execução é realizada na *thread* principal, todas as operações de I/O são delegadas a funções assíncronas e atreladas a um evento. Todas as bibliotecas e APIs do Node utilizam funções assíncronas que recebem *callbacks*.

Porém, um lado negativo muito forte dessa abordagem é que aplicações que possuem operações que fazem uso intensivo da CPU, pois travam a *thread* principal fazendo com que novas requisições demorem para serem respondidas. Portanto, não é recomendado usar Node para aplicações com esse tipo de requisito.

Todo o *back-end* da solução desenvolvida foi implementada em NodeJS, incluindo a lógica de interpretação e transformação dos diagramas.

4.1.2 Typescript

Javascript é uma linguagem orientada a objetos e dinamicamente tipificada, ou seja, os tipos de suas variáveis são definidos em tempo de execução. Apesar de ser definida como uma linguagem orientada a objetos (OO), seus objetos são na verdade definidos através de funções, que são um tipo de objeto na linguagem, e não possui

alguns recursos importantes encontrados em outras linguagens OO, como por exemplo o conceito de interfaces e modificadores de acesso.

Typescript é uma linguagem estaticamente tipificada, orientada a objetos, desenvolvida e mantida pela Microsoft como um projeto de código aberto. Possui todos os recursos de linguagens orientada a objetos e é compilada em Javascript, de forma que todo código Javascript é um código Typescript válido [20]. Por ser estaticamente tipificada é possível verificar erros antes da execução do código, o que facilita o desenvolvimento. Além de tornar mais fácil de estruturar sistemas de médio e grande porte, através da orientação a objetos feita com classes, como já é de costume em linguagens como Java e C#. Por isso foi adotado o uso dessa linguagem, todo código foi escrito em Typescript.

Para realizar o desenho dos diagramas UML, foi utilizada uma biblioteca Javascript chamada JointJS, que permite desenhar diversos tipos de diagramas e customiza-los de acordo com a necessidade do usuário. Também é possível interagir com o diagrama no browser, posicionando seus elementos para melhor visualização. Todos os diagramas gerados pela ferramenta seguem a notação do diagrama de classes UML.

4.2 Projeto e implementação da solução

A ferramenta desenvolvida possui as seguintes funcionalidades básicas: interpretação do XML gerado pelo Astah para extração das informações necessárias dos dois diagramas (modelo de funcionalidades e modelo de interação de serviços), aplicação das regras para gerar os diagramas de arquitetura de serviço e modelo de componentes e exibir os diagramas gerados em formato UML para o usuário.

Objetivando satisfazer esses requisitos e implementar um sistema fácil de manter e evoluir, foram utilizados os seguintes padrões de projeto: *Facade*, *Factory*, *Singleton* e *Model View Controller* (MVC). A figura 4.5 mostra uma visão geral do diagrama de classes da solução e em seguida será discutido cada padrão e como foi utilizado no projeto.

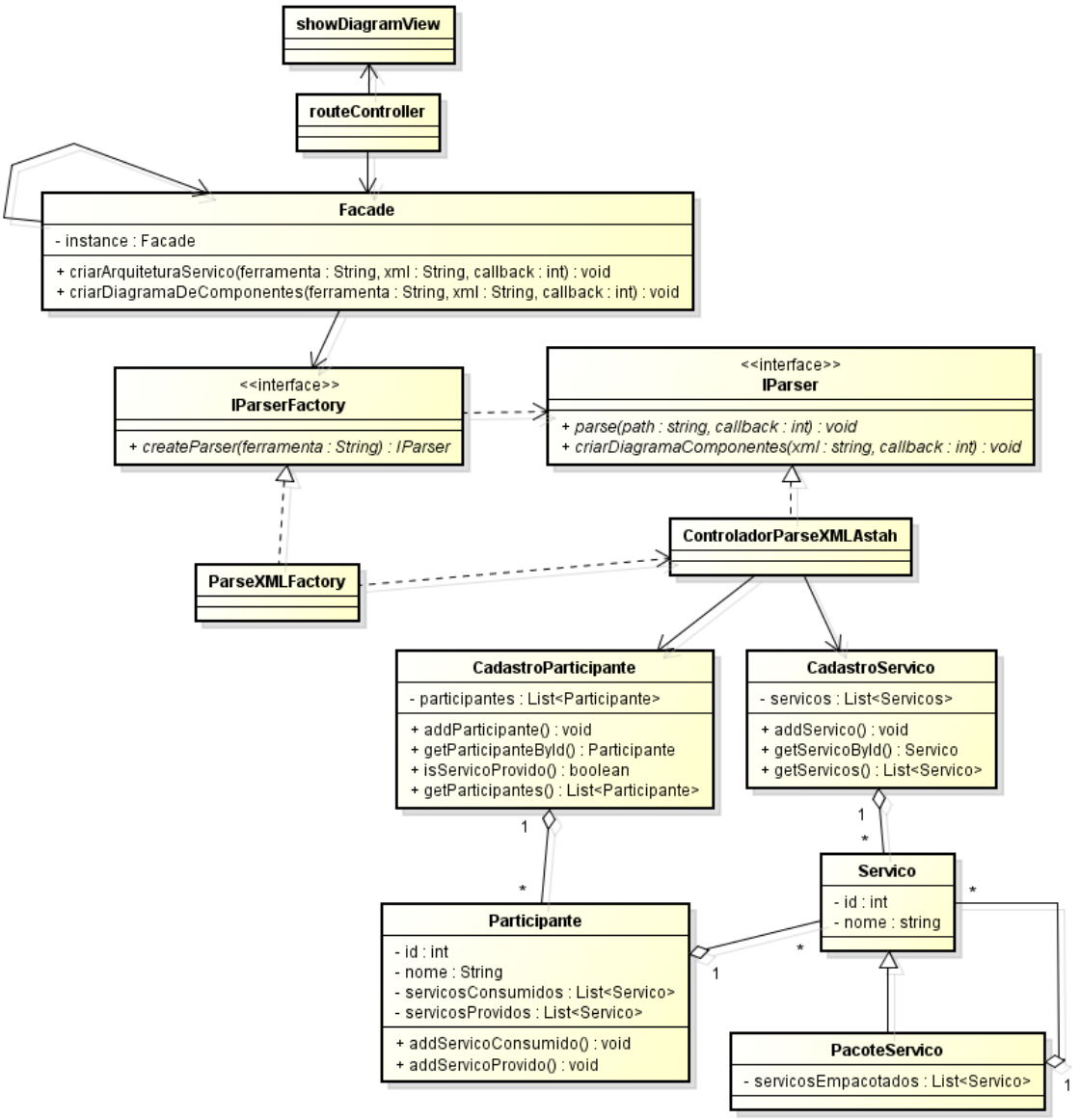


Figura 4.5 – Diagrama de classes da solução proposta.

As entidades básicas da solução são representadas pelas classes “Participante”, “Servico” e sua especialização “PacoteServico”. O primeiro possui um identificador, nome, coleção de serviços consumidos e ofertados. A entidade “Servico” possui um identificador, um nome e uma especialização. No modelo funcional os serviços são agrupados em pacotes que por sua vez são transformados em serviços nos diagramas gerados e cada um contém uma coleção de serviços, por isso foi necessário criar a entidade “PacoteServico” que herda de “Servico”, compartilhando o mesmo tipo; aqui, temos uma instância do padrão de projeto *composite* [22]. A ferramenta não persiste os dados, suas entidades são salvas em memória e manipuladas através das classes “CadastroParticipante” e “CadastroServico”. A classe “ControladorParseAstahXML”, que

implementa a interface "*IParser*", é a responsável por extrair as entidades do XML dado como entrada e gerar a arquitetura de serviço. Todas as funcionalidades do sistema são acessadas pela sua fachada, representada pela classe "*Facade*".

O padrão MVC, já explanado na seção 2.2.2, é implementado com ajuda do framework ExpressJS que já fornece toda a estrutura de roteamento, criação de *webserver* e comunicação entre os componentes *view* e *controller*. No diagrama de classes apresentado acima, a *view* é representada pela casse "*showDiagramaView*", a camada de *controllers* é representado pela classe "*routeController"* e todas as outras classes compõe a camada *model* do padrão.

4.2.1 Padrão *Facade*

O padrão fachada consiste em prover uma interface única de acesso a todo o sistema, agrupando todas as funções utilizáveis publicamente do sistema. Ou seja, todo o sistema é encapsulado dentro de um único objeto [22]. A figura 4.6 demonstra o diagrama do padrão fachada. Apesar de ser simples, é bastante útil para desacoplar o uso do sistema por diversos clientes.

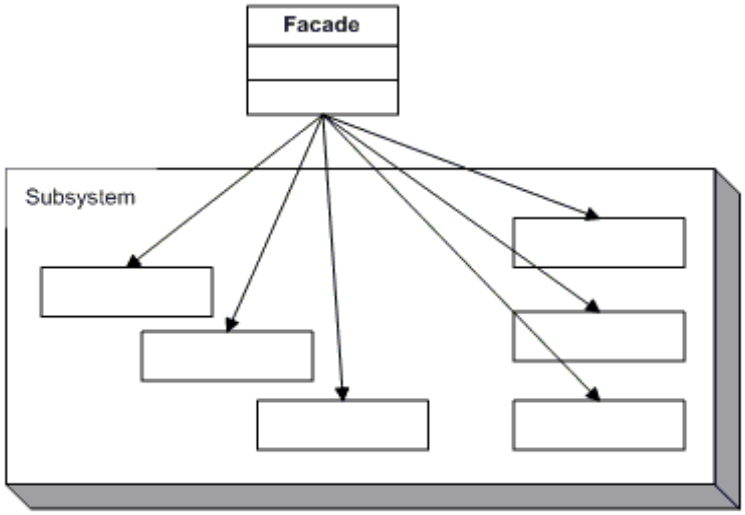


Figura 4.6 – Padrão Fachada [22].

4.2.2 Padrão *Singleton*

A fachada é o único ponto de acesso ao sistema, portanto deve possuir uma única instancia e todos os cliente devem utilizar a mesma instância da fachada. Esse é o objetivo do padrão *singleton*, muitas vezes utilizado em conjunto com o padrão *facade.* A figura 4.7, extraída do diagrama de classes da ferramenta, representa o digrama de classes do padrão.

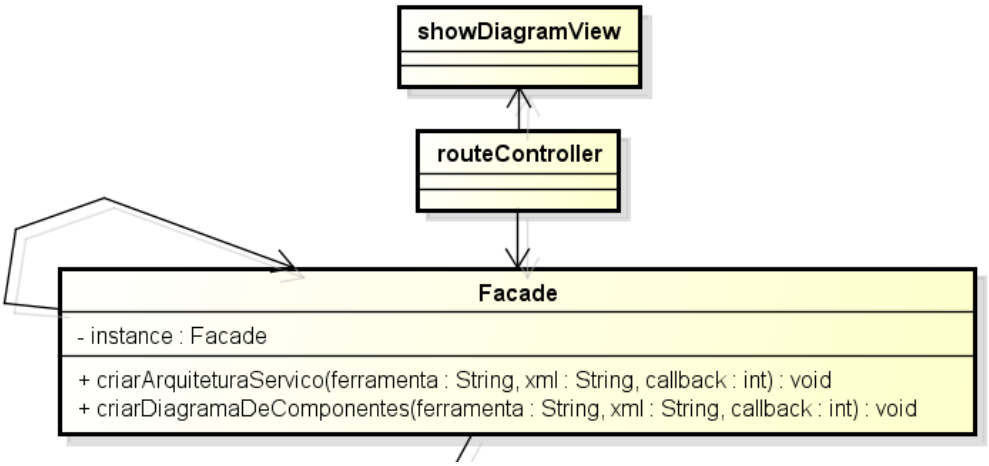


Figura 4.7 – Padrão Singleton.

Para garantir que apenas uma instância de um objeto seja criada, o construtor da classe é declarado privado e um atributo de mesmo tipo da classe é criado, nesse caso o atributo "*instance"* do tipo "*Facade*". Para adquirir uma instancia da classe, é chamado o método público "*getInstance()*" que verifica se o atributo "*instance"* é nulo, instanciando-o caso positivo (primeira vez em que é invocado) ou retornando o atributo já inicializado. A figura 4.8 apresenta o código que implementa essa lógica.

```
import ParserFactory = require('./parser/ParseXMLFactory');
import Parser = require('./parser/IParser');

class Facade{
    private parserFactory : ParserFactory;
    private static instance : Facade;

    constructor(){
        this.parserFactory = new ParserFactory();
    }

    public static getInstance() : Facade{
        if(!Facade.instance){
            this.instance = new Facade();
        }

        return this.instance;
    }

    public criarArquiteturaServico(ferramenta:string,xmlPath:string,callback) : void{
        var parser:Parser = this.parserFactory.createParser(ferramenta);
        parser.parse(xmlPath,callback);
    }
}

export = Facade;
```

Figura 4.8 – Implementação padrão *Singleton* em Typescript.

O código acima faz parte do código da ferramenta, e está escrito em typescript. Ao contrário do Java e similares, o tipo do atributo vem depois de seu nome separado pelo caractere ":". A lógica do padrão é implementada no método estático "*getInstance()*", invocado por todos os clientes da classe.

4.2.3 Padrão *Factory*

O padrão *Factory* ou fábrica delega a criação de objetos ou famílias de objetos a classes especificas que implementam a interface da fábrica, dessa forma a lógica de criação não é exposta a clientes, que referenciam objetos criados através de uma interface comum. Esse desacoplamento entre os clientes e a criação de objetos é a grande vantagem de se utilizar esse padrão. A figura 4.9, extraída do diagrama de classes da ferramenta, demonstra como é o diagrama do padrão *Factory.*

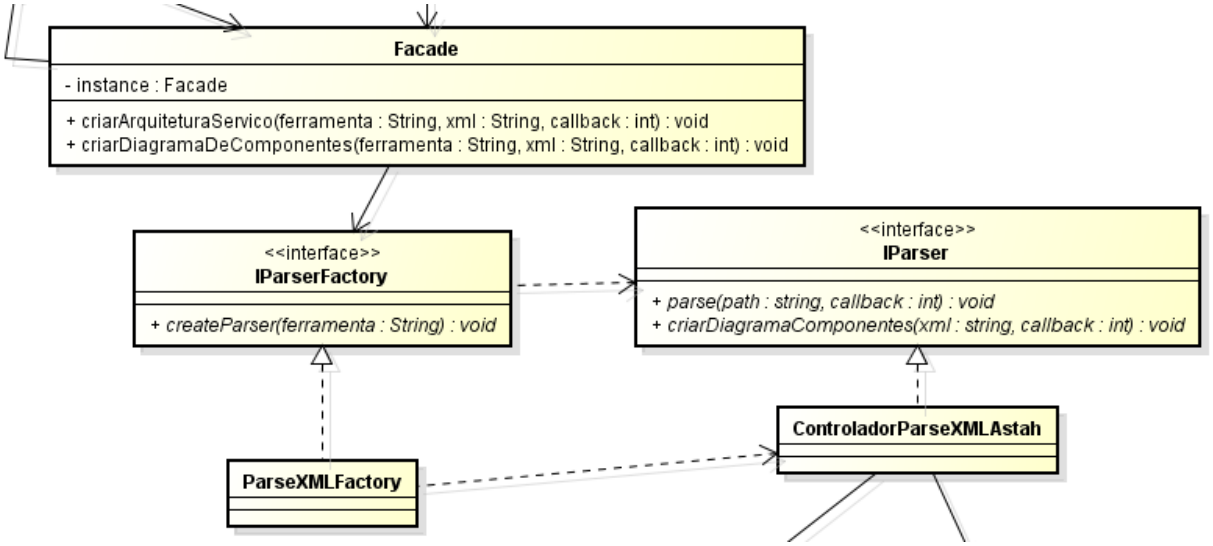


Figura 4.9 – Padrão *Factory*.

Como citado no início do capítulo, cada ferramenta CASE representa seus diagramas de forma diferente, de modo que a lógica de interpretação e extração de dados varia para cada uma. Visando oferecer suporte a diversas ferramentas em versões futuras, foi utilizado o padrão *Factory* com o intuito de facilitar essa evolução. Na versão inicial, apresentada nesse trabalho, a ferramenta oferece suporte apenas para diagramas exportados pelo Astah. A fábrica, definida pela interface "IParserFactory", recebe a identificação de uma ferramenta e instancia a classe correta para fazer a tradução do XML. O cliente, a fachada, só tem conhecimento da *Factory* e da interface "*IParser",* para adicionar suporte a outras ferramentas só é necessário implementar essa interface e adicionar sua criação na fábrica. O cliente não tem conhecimento de quem está executando as operações no XML, sendo possível alterar esse objeto em tempo de execução. Essa é a grande vantagem de se usar o padrão, que permite evoluir o sistema dando suporte a diversas ferramentas de modelagem UML sem afetar o funcionamento de classes já existentes.

Capítulo 5 - Estudo de caso

A demonstração da ferramenta desenvolvida é feita no processo de elaboração de arquitetura de um projeto que consiste em elaborar uma arquitetura orientada a serviços para um aplicativo denominado *SharingShopListApp* detalhado a seguir.

O *SharingShopList* é um aplicativo para dispositivos móveis projetado para as famílias e/ou apartamentos compartilhados. Com ele o usuário pode compartilhar sua lista de compras com quem desejar antes de ir ao supermercado. Assim, é possível saber o que falta em casa e quem fará as compras. Isto economiza tempo e dinheiro. Algumas funcionalidades são:

- Lista de compras compartilhada;
- Criar múltiplas listas;
- Adicionar quantidades e descrições;
- Compartilhar listas por texto, e-mail, WhatsApp, etc;
- Marcar os itens da lista como "comprado" ou "no carrinho";
- Editar os itens na lista;
- Gerar notificação *push* quando uma lista for alterada;
- Ganhar tempo usando as sugestões assim que começar a digitar o nome de um produto;
- Receber sugestões personalizadas de produtos;
- Classificação automática;
- Comissão por compra realizada em site parceiro;
- Níveis de acesso para usuários que compartilham a lista;

Assim que o usuário começa a digitar o nome de um produto, o *SharingShopList* mostra sugestões que ele pode adicionar de forma rápida à sua lista de compras. Além disso, ele já vem com várias recomendações personalizadas de compras para o usuário. Todos os produtos colocados no carrinho estarão disponíveis futuramente como sugestões, para poupar tempo quando estiver montando a lista de compras. A figura 4.10 mostra o diagrama de casos de uso do sistema.

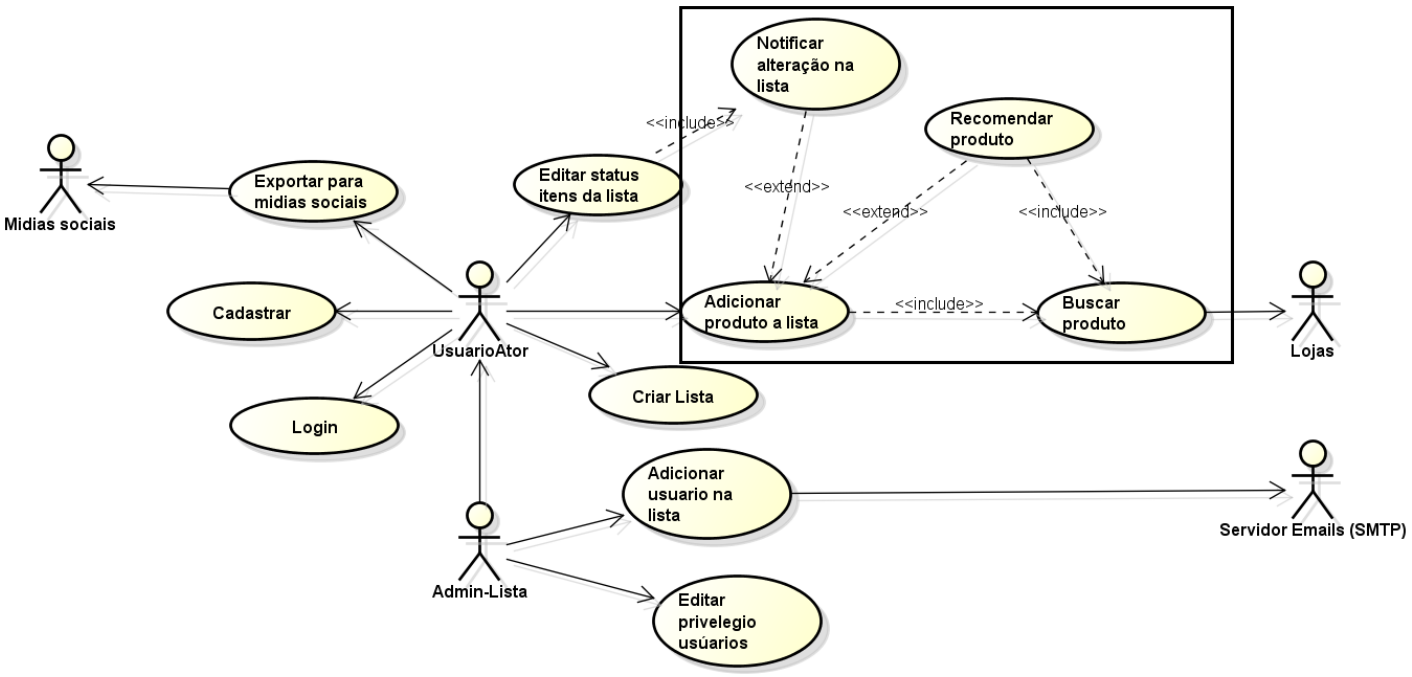


Figura 4.10 – Diagrama de casos de uso do *SharingShopListApp.*

O aplicativo possui integração com mídias sociais, faz busca de produtos em sistemas de lojas e se comunica com o usuário utilizando servidores de e-mail, proporcionando um ótimo exemplo para se construir uma arquitetura SOA. Para manter o foco na demonstração da ferramenta desenvolvida, são exemplificadas apenas as tarefas, do processo apresentado no capítulo 3, que envolvem seu uso. Uma demonstração completa do processo pode ser encontrada no trabalho de BRAGA, V [6].

5.1 Guia para o uso da ferramenta

O primeiro passo é a criação do modelo de funcionalidades, que pode ser feito no Astah criando um diagrama de caso de uso conforme mostra a figura 4.11.

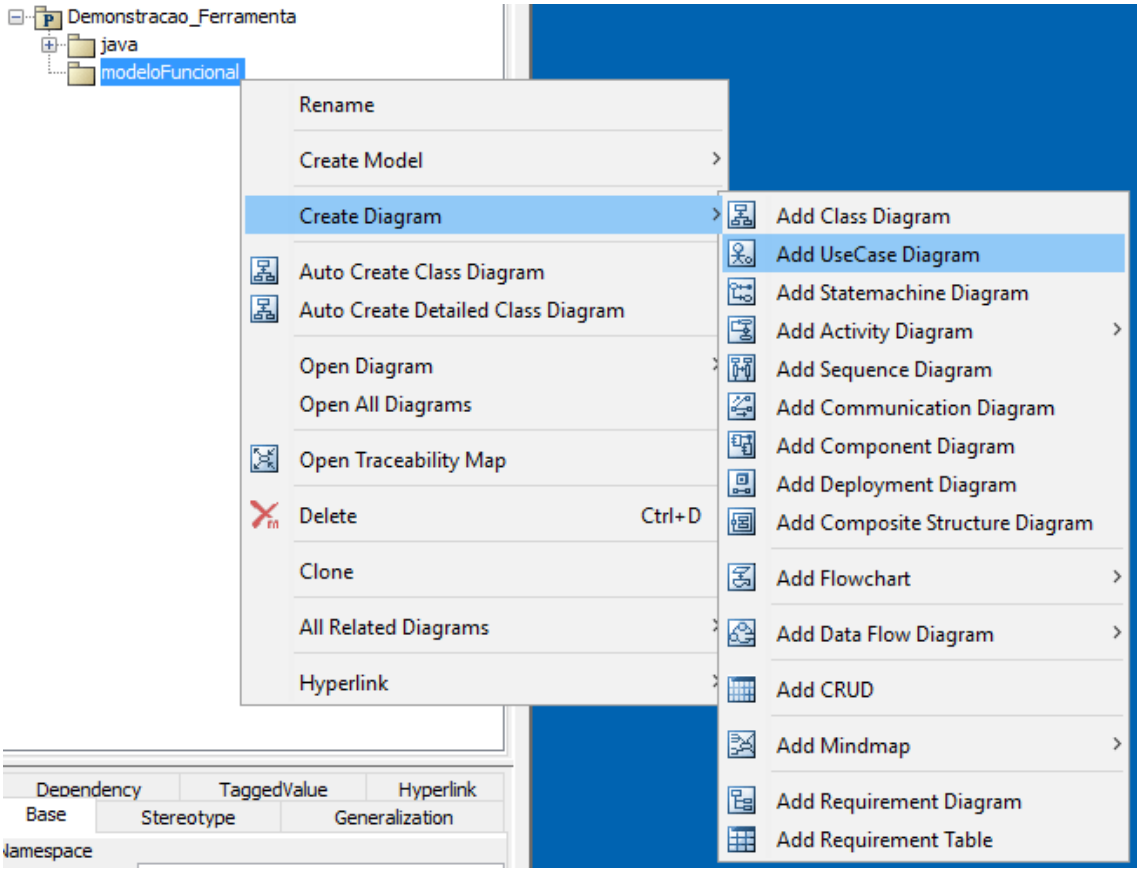


Figura 4.11 – Criação de diagrama de casos de uso.

Ao iniciar um novo projeto no Astah, um *workspace* contendo apenas a pasta java é criado. Diagramas e elementos são salvos nele, e quando a função de exportar XML é executada, todos os elementos e diagramas presentes no *workspace* são escritos no arquivo gerado. Para utilizar a ferramenta é necessário agrupar os diagramas criados em pacotes, o que torna possível extrair suas informações corretamente. É necessário criar o pacote “modeloFuncional” e dentro desse criar o diagrama de caso de uso, como mostrado na figura 4.11. A ferramenta irá procurar especificamente por um pacote com esse nome, e entenderá que todos os elementos presentes nele fazem parte do modelo funcional.

A elaboração do modelo funcional consiste em agrupar os casos de uso de acordo com funcionalidades relacionadas; os que estão presentes em um mesmo pacote fazem parte do mesmo serviço. A figura 4.12 mostra o MF criado a partir dos casos de uso apresentados.

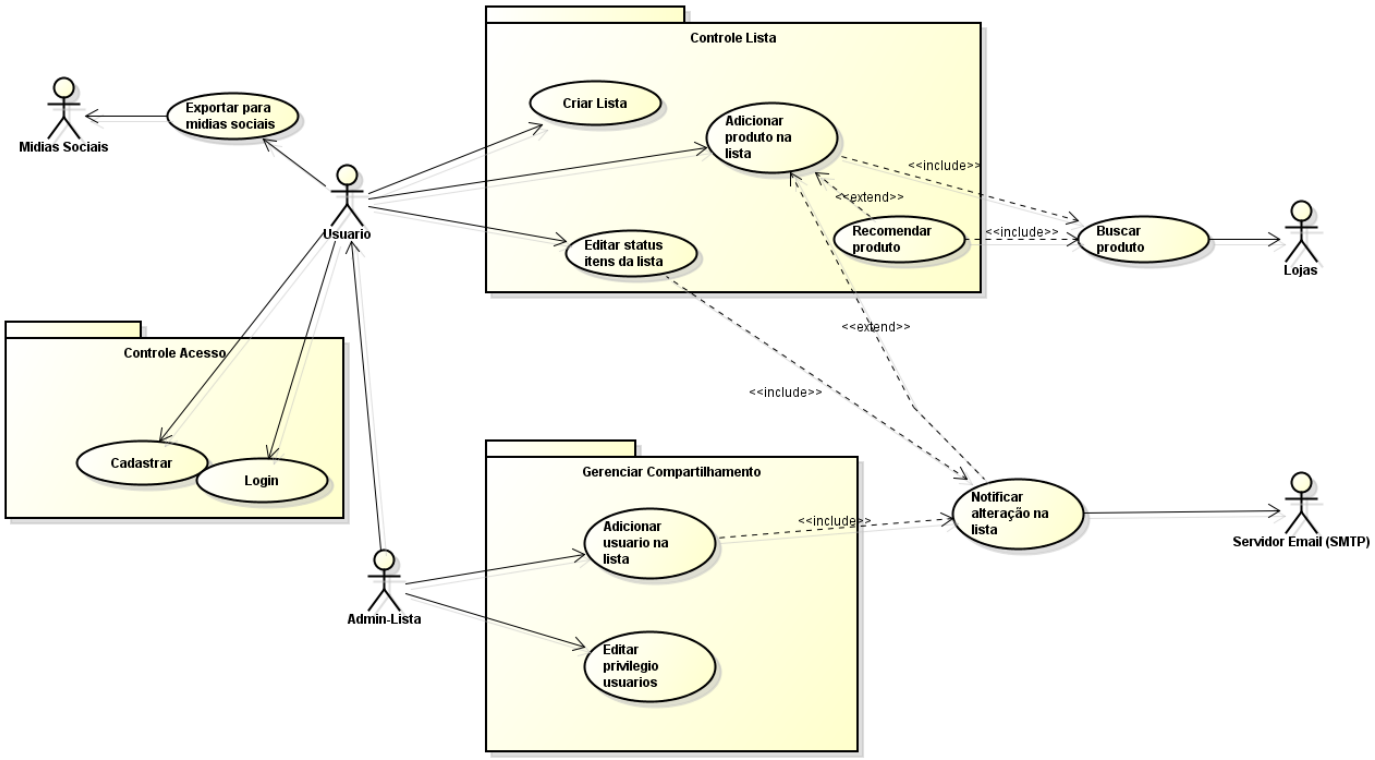


Figura 4.12 – Modelo funcional do *SharingShopListApp*.

Basicamente casos de uso referentes à lista foram agrupados no pacote “Controle Lista”, os que têm função de controlar acesso ao sistema são agrupados no pacote “Controle Acesso” e aqueles que têm o objetivo de compartilhar a lista estão presentes no pacote “Gerenciar Compartilhamento”. Casos de uso que fazem comunicação com outros atores não foram agrupados em nenhum pacote, pois são serviços ofertados por sistemas terceiros.

O próximo passo é exportar o diagrama criado em XML para geração automática da arquitetura de serviço. A figura 4.13 demonstra como isso é feito no Astah.

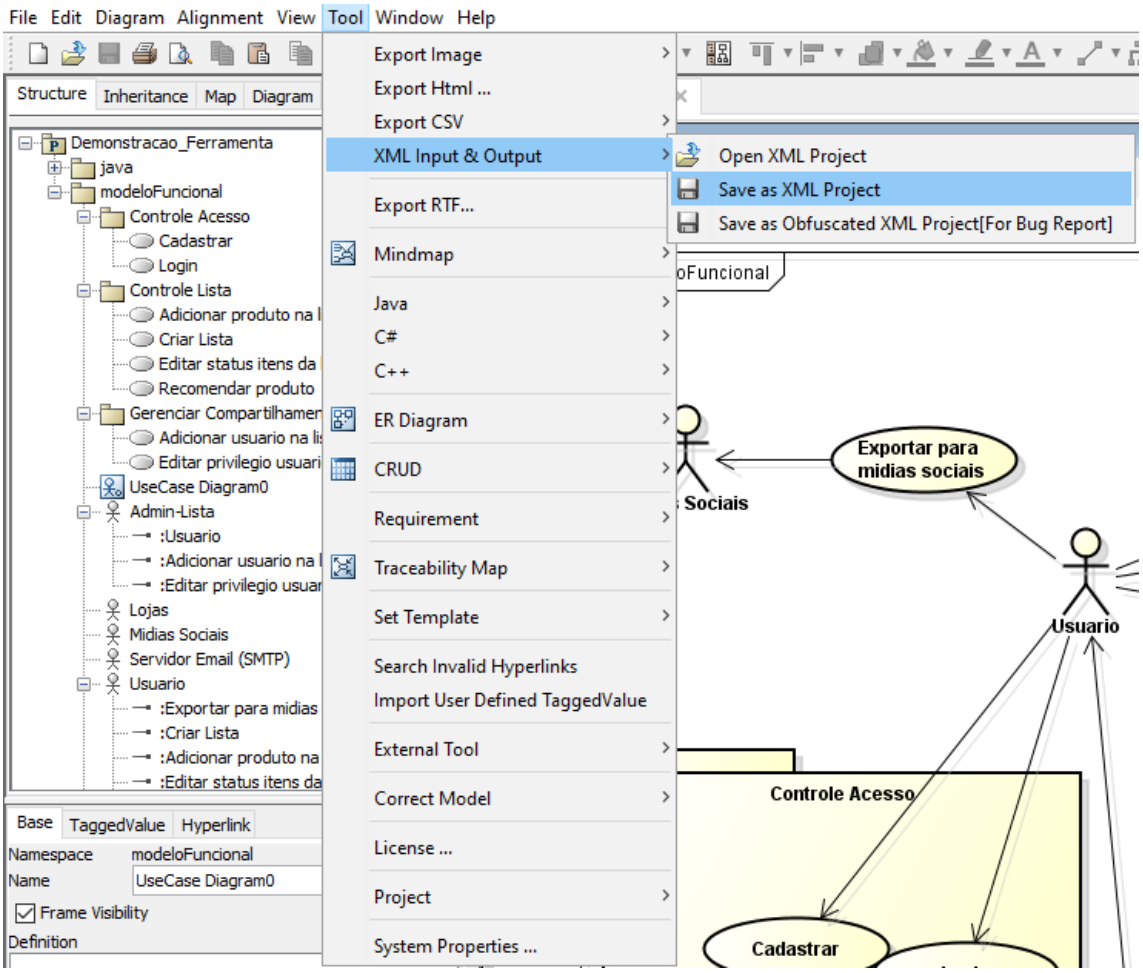


Figura 4.13 – Exportar diagramas para XML.

Após selecionar a opção "*Save as XML Project*", basta escolher o local para salvá-lo. É interessante perceber a mudança no *workspace* no lado esquerdo da figura 4.13, todos os elementos criados são inseridos dentro da pasta "modeloFuncional", inclusive o próprio diagrama.

Com o XML em mãos basta acessar o site da ferramenta, fazer o upload do arquivo e o diagrama de arquitetura de serviço é gerado, mostrado na figura 4.14.

.

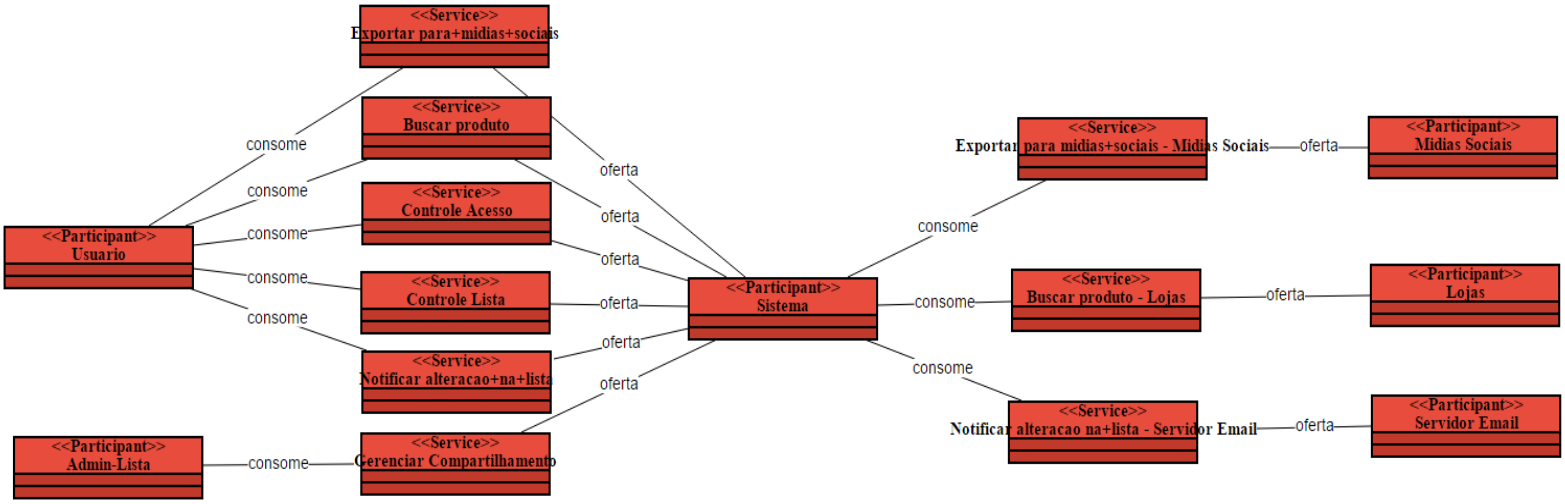


Figura 4.14 – Arquitetura de Serviço gerado pela ferramenta.

Todos os casos de uso presentes em pacotes foram mapeados para contratos de serviço e o sistema consome e oferece todos serviços isolados oferecidos pelos participantes externos, disponibilizando-os para o participante “Usuário” e “Admin-lista”. Porém ainda não foram identificados os serviços de entidades e como é sua interação com os serviços já identificados, para extrair essas informações é necessário construir o modelo de interação de serviço. As figuras 4.15 mostra um exemplo do MIS.

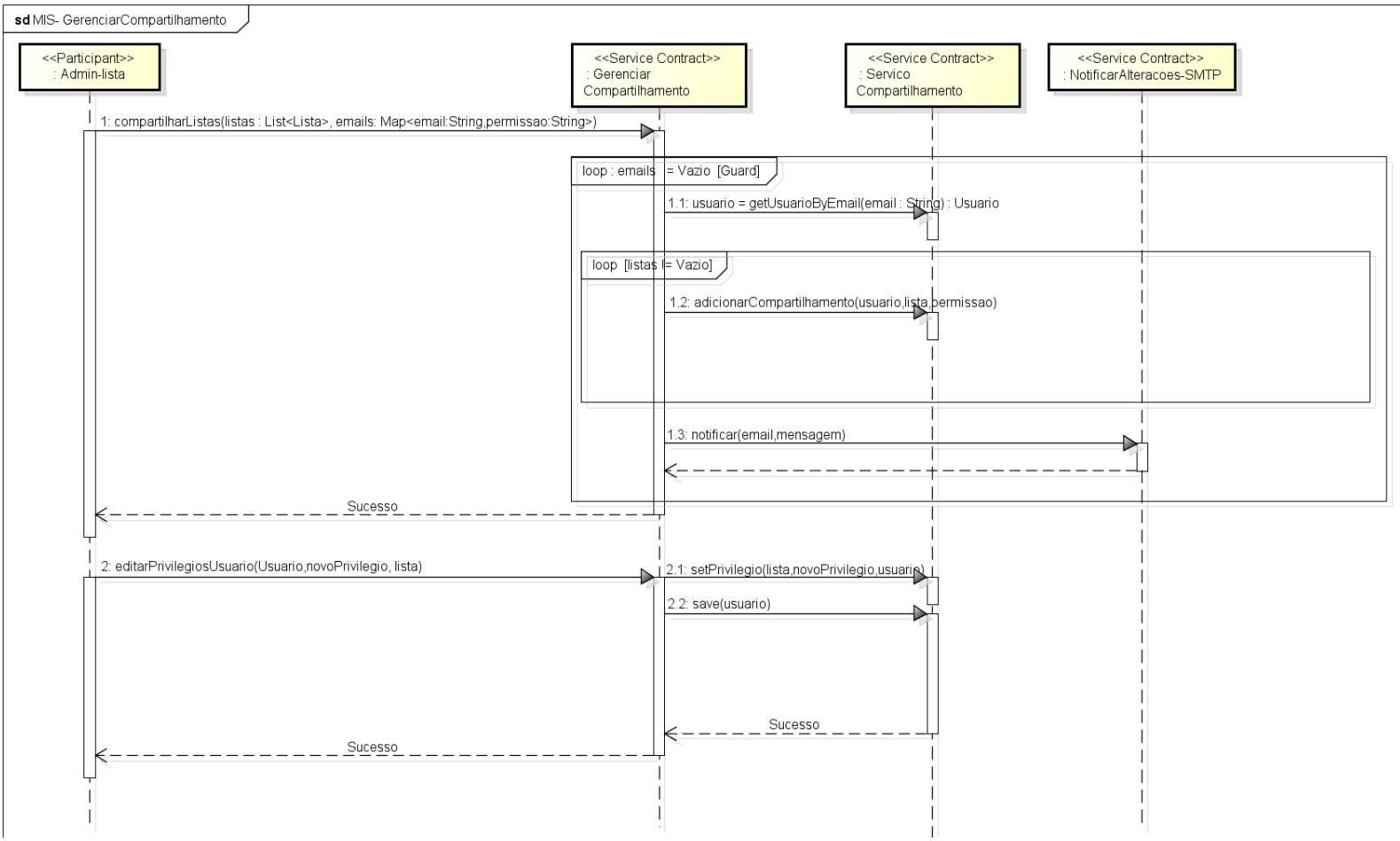


Figura 4.15 – Modelo de Interação de Serviços.

A partir desse modelo é possível identificar o serviço de entidade “Servico Compartilhamento” que não estava presente no modelo funcional. E é possível ver quem interage com esse novo serviço. Esse diagrama pode ser criado no Astah como um diagrama de sequência. Para que a ferramenta consiga gerar o modelo de componentes, é necessário que todos os diagramas de interação de serviço do sistema estejam contidos na pasta “modeloFuncional” no *workspace.* Dessa forma, quando o arquivo XML for importado para a ferramenta, todos os diagramas de interação serão analisados para extrair como os serviços interagem e os serviços de entidades não identificados previamente. A figura 4.16 mostra o modelo de componente de serviço gerado automaticamente.

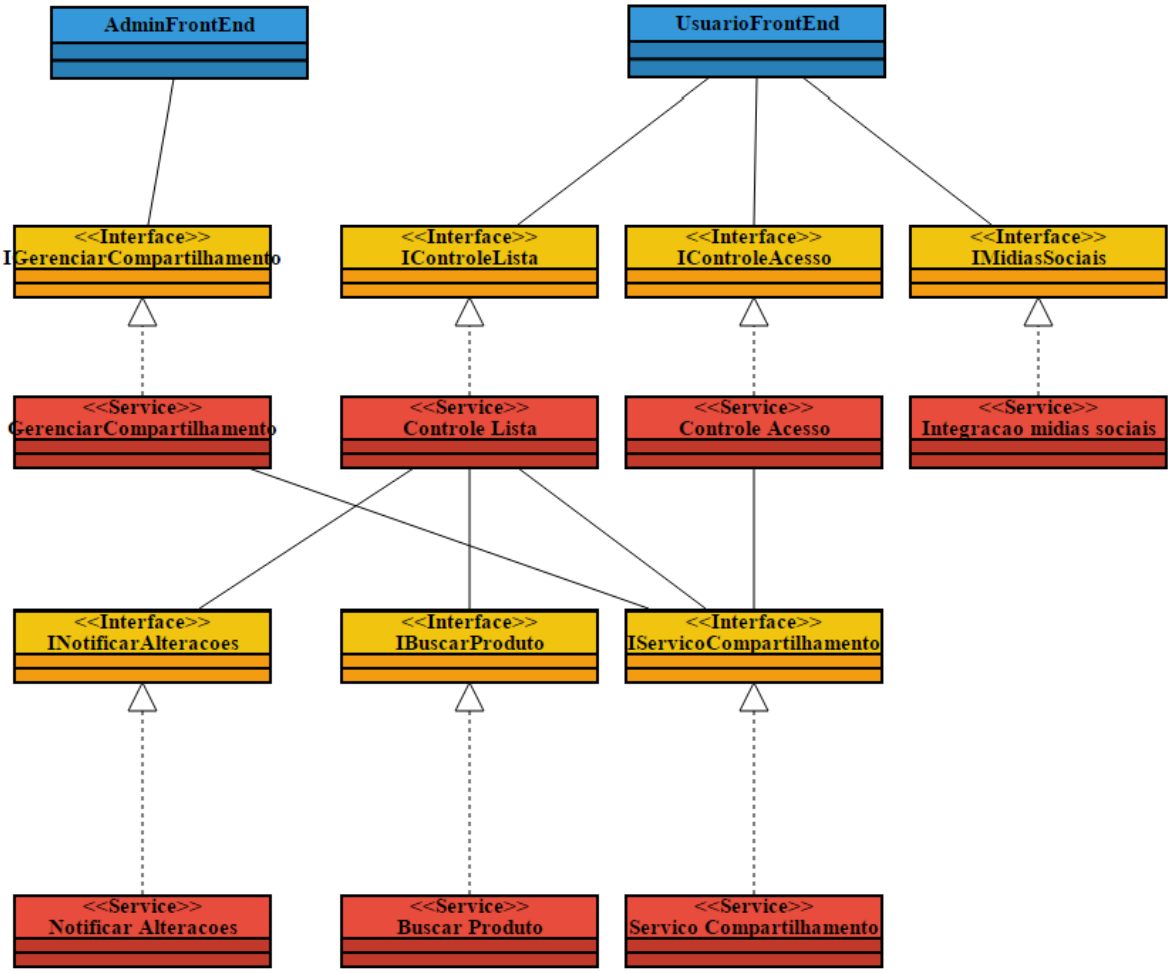


Figura 4.16 – Modelo de componentes de Serviços.

Com esse diagrama é possível ter uma visão macro de como os componentes do sistema interagem para atender os requisitos. É possível também ter uma separação entre o front-end e o back-end da aplicação. Assim conclui-se o uso da ferramenta desenvolvida, que automatiza a criação desses dois diagramas do processo proposto.

Capítulo 6 – Conclusão

A preocupação com a grande quantidade de falhas em projetos de software, baseada em dados que vem sendo apresentado em estudos como o CHAOS Report [13], ao longo dos anos, alertou a necessidade de se repensar no processo de desenvolvimento de sistemas. Isso vem resultando no surgimento de diversos estudos para propor alternativas, novas práticas e melhorias nas metodologias de desenvolvimento.

Na tarefa de análise e projeto de sistemas, mais especificamente, foram propostos diversos processos de elaboração, padrões e estilos arquiteturais, como o SOA, que apesar de ser promissor e possuir diversas vantagens, como o foco em reuso de componentes, ainda não possui um apoio ferramental para elaborar esse tipo de arquitetura.

Este trabalho se baseou no processo de projeto arquitetural orientado a serviços elaborado por Braga [6]; o objetivo foi implementar uma ferramenta que facilite a utilização do processo proposto, através da automatização de algumas tarefas. Em sua versão inicial, apresentada nesse trabalho de graduação, foram automatizadas as tarefas Identificar Serviços e Identificar Componentes, gerando os artefatos: Arquitetura de Serviço e Modelo de Componentes de Serviço.

6.1 Principais Contribuições

Com uso da ferramenta desenvolvida é possível agilizar a elaboração do processo, pois tarefas mecânicas como aplicação das regras para geração dos diagramas propostos são feitas automaticamente e reduzem o risco de ocorrer erros. Sobre a ferramenta é importante destacar:

- É uma aplicação web – Foi desenvolvida como uma aplicação web, que caso seja continuada e hospedada, pode ser utilizada de qualquer local e usa o conceito de software como um serviço (saas, do inglês *software as a service*).
- Padrões arquiteturais – Implementada seguindo o padrão arquitetural MVC e diversos padrões de projeto, visando uma fácil evolução do sistema, onde já é possível prever mudanças futuras.

- Automatização do processo – Através da implementação de regras sistemáticas e bem definidas é possível realizar a transformação dos diagramas Modelo Funcional e Modelo de Interação de Serviço no diagrama de Arquitetura de Serviço e Modelo de Componentes de Serviço, utilizando notação SoaML.

6.2 Trabalhos Relacionados

Este trabalho se baseou no processo de desenvolvimento de software centrado em arquitetura, baseado no RUP, orientada a serviços e dirigido a modelos desenvolvido por Braga [6] e revisado por Peixoto [23].

Através do processo sistemático proposto, é possível resolver problemas práticos encontrados no desenvolvimento de sistemas, como: a integração de aplicações, o acoplamento entre *front-end* e *back-end*, e o desalinhamento de expectativas entre os stakeholders. Além do foco em reutilização de software, prover uma arquitetura modular e estável e ser prático e fácil de se utilizar [6].

6.3 Trabalhos Futuros

O escopo da ferramenta desenvolvida para esse trabalho foi implementar as regras de transformação dos diagramas citados. Um trabalho futuro imediato é a geração automática dos demais artefatos produzidos pelo processo apresentado em [6], incluindo, por exemplo, o projeto detalhado da arquitetura, do projeto do *front-end* e do *back-end* da aplicação.

Na versão atual da ferramenta não é possível editar os artefatos gerados e nem importá-los em outras ferramentas CASE, o usuário pode apenas organizar a posição dos elementos UML nessa versão inicial.

Em trabalhos futuros seria interessante dar suporte a uma melhor integração com ferramentas CASE, exportando os artefatos gerados para serem consumidos por elas. No entanto, essa abordagem gera o mesmo problema encontrado na importação do XML, cada ferramenta o espera em uma formatação diferente, sendo necessário

exportar os diagramas nos formatos esperados por cada ferramenta que se deseja dar suporte.

Uma outra alternativa, seria transformar a solução desenvolvida em uma ferramenta CASE com suporte a SoaML. Dessa forma, todos os diagramas do processo poderiam ser feitos e gerados automaticamente dentro do mesmo ambiente. Uma funcionalidade extra interessante, seria ter um ambiente colaborativo em que diagramas pudessem ser criados e editados em tempo real por diversos usuários simultaneamente.

Bibliografia

[1]. BASS, L.; CLEMENTS, P. & KAZMAN, R. **Software Architecture in Practice**. 2 ed. Sei series in software engineering, Addison-Wesley, 2003.

[2]. FOWLER, M. **Patterns of enterprise application architecture**. Addison-Wesley, 2002.

[3] **Architectural Patterns.** 2001. Disponível em: http://pubs.opengroup.org/architecture/togaf7-doc/arch/p4/patterns/patterns.htm. Acesso em: 05/12/2015.

[4] **Pattern**-*Oriented Software Architecture: A System of Patterns*, by F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal, John Wiley and Sons, 1996, ISBN 0-471-95869-7.

[5] C. Alexander: The Timeless Way of Building. Oxford University Press. 1979.

[6] BRAGA, V. **Um processo para projeto arquitetural de software dirigido a modelos e orientado a serviços.** Dissertação de Mestrado. Centro de Informática (CIn), Universidade Federal de Pernambuco, 2011

[7] **Layered Application Guidelines.** Disponível em: https://msdn.microsoft.com/en-us/library/ee658109.aspx. Acessado em: 10/12/2015.

[8] **Padrões de Interação com o Usuário**. Disponível em: http://cin.ufpe.br/~if718/. Acessado em: 10/12/2015

[9] PAPAZOGLOU, M. P. & HEUVE, W.-J. **Service oriented architectures: approaches, technologies and research issues**. Springer-Verlag, 2007. Disponível em: <ftp://ftp-sop.inria.fr/members/Didier.Parigot/PDFarticle/francesco/Papazoglou2007a.pdf>. Acesso em: 18/12/2015.

[10] MAHMOOD, Z. **The promise and limitations of service oriented architecture**. International Journal of Computers, Issue 3, Volume 1, 2007.

[11] EARL, Thomas. **Service-Oriented Architecture: Concepts,Technology and Design.** Prentice HALL PTR, 12/08/2005. 792 p.

[12] RIBEIRO, Heberth Braga Gonçalves; ALVES, V.; Garcia, C. Vinicius; Álvaro, Alexandre; Lucrédio, Daniel; Almeida, S. Eduardo; Meira, L. R. Silvio. **An Assessment on Technologies for Implementing Core Assets in Service Oriented Product Lines**. In: IV

Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), 2010, Salvador.

[13] **CHAOS Summary** 2015. Disponível em: http://www.standishgroup.com/newsroom/ chaos_2015.php ,acessado em: 15/10/2015.

[14] GALORATH, **D. D. Software total ownership costs: Development is only job one**. 2014. Disponível em http://galorath.com/wp-content/uploads/2014/08/software_total_ownership_costs-development_is_only_job_one.pdf. Acessado em: 12/10/2015.

[15] **SOA – Service Oriented Architecture.** Aula 20. Disponível em: http://cin.ufpe.br/~if718/. Acessado em: 13/01/2016

[16] TEXEIRA, PEDRO. **Professional NodeJS Building javascript based scalable software**. Wrox, 2012. ISBN 978-1118185469.

[17]**Node Packet Manager Repository.** Disponível em https://www.npmjs.com/. Acessado em: 13/01/2016.

[18] **NodeJs.** Disponível em: https://nodejs.org/en/. Acessado em: 13/01/2016.

[19]**NodeJs Perfomance Event Loop Monitoring.** Disponível em: https://strongloop.com/strongblog/node-js-performance-event-loop-monitoring/. Acessado em 13/01/2016.

[20]**TypeScript.** Disponível em: https://en.wikipedia.org/wiki/TypeScript. Acessado em: 12/01/2016.

[21]**Typescriptlang.** Disponível em: http://www.typescriptlang.org/. Acessado em: 12/01/2016.

[22] FREEMAN, Eric; FREEMAN, Elizabeth; SIERRA, Kathy; BASS, Bates.**Head First Design Patterns.** O'Reilly,2004.

[23] PEIXOTO, Diogo. **Integrando SOA e MDD ao RUP.** Trabalho de Graduação. Centro de Informática (CIn), Universidade Federal de Pernambuco, 2011

Assinaturas

Trabalho de graduação: Síntese de Arquitetura Orientada a Serviços a partir de casos de uso.

Aluno: Victor Hugo Silva do Nascimento

Orientador: Augusto Alves Sampaio

Universidade Federal de Pernambuco

Graduação em Ciência da Computação

Centro de Informática

2015.2

_____

Augusto Alves Sampaio

Orientador

_____

Juliano Lyoda

Avaliador